## 课程介绍

**1、讲师自我介绍**

（1）中华石杉

（2）BAT以及一线互联网公司出身，多年的大数据系统研发和架构经验，精通大数据技术栈，包括Hadoop，Spark，Storm，Hive，HBase，ZooKeeper，Elasticsearch，数据仓库，数据挖掘，大型Java系统架构，等等

（3）之前有过丰富的大型视频课程录制经验（spark），以及丰富企业内部培训和技术讲座、技术分享的经验

（4）Elastissearch技术，第一次合作

**2、讲课风格介绍**

（1）不喜欢ppt讲课，照本宣科，一本正经，对着文字和图片干讲 ----> txt文本边讲边写，画图软件手工画图

（2）不喜欢学术型的讲课，一板一眼，像老师在上课 ----> 像朋友之间聊天一样，口语化，轻松话，用大白话式的方法来讲解复杂的技术

（3）要录，就录最好的课程，知识体系全面；原理剖析透彻；全程实战演练驱动；搭配真实项目实战；讲解就业知识

**3、课程内容介绍**

\*\* （1）核心知识篇 \*\*

课程特点：

（1）使用最新Elasticsearch 5.2版本讲解，市面上的书籍和视频几乎都停留在2.x版本

（2）深入浅出ES核心工作原理，全部手工画图讲解，完全不同于市面上已有视频的PPT讲解

（3）涵盖Elasticsearch所有核心知识点，系统化，体系完整详细，有一定深度，包括完整Java开发示范

（3-1）全面的知识体系，包括了工作原理，文档管理，索引管理，搜索，聚合分析，分词，数据建模，Java API等知识

（3-2）知识足够深入和细节，完全秒杀市面上已有的书籍和视频，比如index segment merge原理，乐观锁并发控制，索引别名与零停机，相关度评分算法与定制，近似聚合算法，doc values与fielddata机制原理，父子关系数据建模，Java API执行scroll search等各种复杂操作，等等

（4）全程每讲必练，大量的案例实战和上机实验，实战出真知，实战中学知识，没有任何一讲是干讲ppt的

（5）包含一个实战项目，运用学到的知识，开发一个小型门户网站的搜索引擎和数据分析系统，运用ES几乎所有的核心知识，不像市面上的demo项目

（6）课程学完之后，学员可以掌握es所有核心知识点，理解es核心原理，而且能够熟练动手操作所有学到的知识和功能，并且能够掌握ES集群的基本部署，并且基于Java开发一个适用于中小型企业的搜索引擎以及数据分析系统，达到学完即可上手到中小型项目中使用的程度

\*\* （2）高手进阶篇 \*\*

课程特点：

（1）使用最新Elasticsearch 5.2版本讲解

（2）包含市面上几乎没有的所有Elasticsearch高级知识点：包含地理位置搜索与聚合分析，term vector，suggester search，搜索模板定制，query执行剖析，数十种最全面的聚合分析，span query，shard分配定制，es插件开发，等等，高级的知识点，这些知识点，市面上已有的书籍或视频几乎都没有

（3）全程每讲必练，大量的案例实战和上机实验

（4）包含一个复杂实战项目，运用学到的知识，开发一个复杂的基于地理位置的智能餐厅app的搜索引擎和数据分析系统，运用ES从核心篇到高级篇的所有高阶知识点

（5）课程学完之后，学员可以掌握es从核心到高阶的所有知识点，掌握完整的有深度的es知识体系，同时能够动手操作所有的知识点和功能，最后通过项目实战，能够在中小型公司中，基于Java开发一个可以基于地理位置进行搜索的高级搜索引擎，以及使用复杂聚合操作进行分析的高级实时数据分析系统

\*\* （3） 大型集群运维优化篇 \*\*

课程特点：

（1）最全面的Elasticsearch运维、管理、调优、故障处理的知识体系：企业级监控体系的搭建，企业级集群部署，集群日常管理策略，集群版本升级方案，集群基准压测方案，集群数据的备份和恢复，系统核心配置参数，性能调优方案，故障处理方案

（2）全程每讲必练，大量上机实验，所有的运维、管理、部署、优化，全部上机实验

（3）从零开始，逐步搭建出一个大型可扩展、高性能、监控体系完善、管理体系健全的分布式集群

（4）学完课程之后，学员除了可以开发复杂的es搜索/分析系统之外，还可以掌握在任何一个公司里，从零开始搭建一个分布式的大型es集群，并制定完善的监控，运维，管理，优化等方案

\*\* （4）大型项目架构篇 \*\*

课程特点：

（1）涵盖Elasticsearch目前最核心的两个应用领域，垂直搜索引擎，实时数据分析

（2）开发出2个企业级的大型复杂项目，是完全真实的大型企业项目，电商搜索引擎，电商实时数据分析平台

（2-1）大型电商搜索引擎，包括了真正复杂的大型企业，大型项目的商业级搜索引擎架构，包括了检索、数据更新、排序、分词、query分析等各个核心模块，同时架构上实现了复杂的缓存机制，热启动机制，防雪崩机制，自动降级高可用机制，等等

（2-2）大型电商实时数据分析平台，完整、复杂而且大型的电商数据分析，包括了完善的数据分析指标体系（运营指标，流量指标，销售转化指标，客户价值指标，商品指标，营销指标，风险控制指标，市场竞争指标），一站式构建出复杂的，企业级的，电商领域数据分析平台

（2-3）之所以要单独拉出一篇做大型项目实战，是因为，之前几篇讲的项目，多是架构较为简单，业务也不复杂的项目，主要适用于中小型公司，而且那两个项目主要是集中在运用ES的技术本身来开发出需要的功能来（搜索/分析）。这一篇讲解的项目，重点是采用大公司的大型复杂项目作为背景，让同学可以掌握基于ES技术的大型项目架构能力，达到架构师的水平。比如说大型电商搜索引擎，主要运用ES来实现，但是ES之外还有大型系统的复杂架构需要讲解。还有大型的电商实时数据分析平台，主要特点是业务繁琐而且复杂，需要基于ES构建出大型的数据分析平台架构。

（3）学完课程之后，学员在之前课程中掌握了es的企业级集群运维管理，复杂搜索引擎和数据分析引用开发的技术基础之上，现在可以运用所学知识，结合电商的领域知识，开发出业务真实而且复杂的，大型的搜索、分析等电商系统以及相关架构，彻底掌握运用ES和ELK相关技术栈在中大型企业中，开发大型项目架构的经验和能力

\*\*（5）ELK深入浅出篇 \*\*

课程特点：

（1）电商系统日志检索平台，采用ELK技术栈，会详细讲解logstash和kibana两个技术，包括logstash的插件机制，监控方案，大规模扩展方案，升级方案，性能调优方案，kibana的可视化展现方案，同时讲解如何使用ELK技术栈开发出大规模日志存储与检索平台

（2）最后，需要真正深入重点讲解logstash和kibana两门技术，并结合ES技术，采用ELK技术栈，实现大型企业级的日志采集和检索平台。

（3）学完课程后，学员应该可以彻底精通ELK技术栈，并能够使用ELK技术栈快速搭建日志检索平台，以及数据可视化平台

# 基础入门篇

## 用大白话告诉你什么是 Elasticsearch

Elasticsearch，分布式，高性能，高可用，可伸缩的搜索和分析系统

1、什么是搜索？

2、如果用数据库做搜索会怎么样？

3、什么是全文检索、倒排索引和Lucene？

4、什么是Elasticsearch？

**1、什么是搜索？**

百度：我们比如说想找寻任何的信息的时候，就会上百度去搜索一下，比如说找一部自己喜欢的电影，或者说找一本喜欢的书，或者找一条感兴趣的新闻（提到搜索的第一印象）

百度 != 搜索，这是不对的

垂直搜索（站内搜索）

互联网的搜索：电商网站，招聘网站，新闻网站，各种app

IT系统的搜索：OA软件，办公自动化软件，会议管理，日程管理，项目管理，员工管理，搜索“张三”，“张三儿”，“张小三”；有个电商网站，卖家，后台管理系统，搜索“牙膏”，订单，“牙膏相关的订单”

搜索，就是在任何场景下，找寻你想要的信息，这个时候，会输入一段你要搜索的关键字，然后就期望找到这个关键字相关的有些信息

**2、如果用数据库做搜索会怎么样？**

做软件开发的话，或者对IT、计算机有一定的了解的话，都知道，数据都是存储在数据库里面的，比如说电商网站的商品信息，招聘网站的职位信息，新闻网站的新闻信息，等等吧。所以说，很自然的一点，如果说从技术的角度去考虑，如何实现如说，电商网站内部的搜索功能的话，就可以考虑，去使用数据库去进行搜索。

1、比方说，每条记录的指定字段的文本，可能会很长，比如说“商品描述”字段的长度，有长达数千个，甚至数万个字符，这个时候，每次都要对每条记录的所有文本进行扫描，懒判断说，你包不包含我指定的这个关键词（比如说“牙膏”）

2、还不能将搜索词拆分开来，尽可能去搜索更多的符合你的期望的结果，比如输入“生化机”，就搜索不出来“生化危机”

用数据库来实现搜索，是不太靠谱的。通常来说，性能会很差的。



**3、什么是全文检索和Lucene？**

（1）全文检索，倒排索引

（2）lucene，就是一个jar包，里面包含了封装好的各种建立倒排索引，以及进行搜索的代码，包括各种算法。我们就用java开发的时候，引入lucene jar，然后基于lucene的api进行去进行开发就可以了。用lucene，我们就可以去将已有的数据建立索引，lucene会在本地磁盘上面，给我们组织索引的数据结构。另外的话，我们也可以用lucene提供的一些功能和api来针对磁盘上额

**4、什么是Elasticsearch？**

## Elasticsearch 的功能、适用场景以及特点介绍

1、Elasticsearch的功能，干什么的

2、Elasticsearch的适用场景，能在什么地方发挥作用

3、Elasticsearch的特点，跟其他类似的东西不同的地方在哪里

**1、Elasticsearch的功能**

（1）分布式的搜索引擎和数据分析引擎

搜索：百度，网站的站内搜索，IT系统的检索

数据分析：电商网站，最近7天牙膏这种商品销量排名前10的商家有哪些；新闻网站，最近1个月访问量排名前3的新闻版块是哪些

分布式，搜索，数据分析

（2）全文检索，结构化检索，数据分析

全文检索：我想搜索商品名称包含牙膏的商品，select \* from products where product\_name like "%牙膏%"

结构化检索：我想搜索商品分类为日化用品的商品都有哪些，select \* from products where category\_id='日化用品'

部分匹配、自动完成、搜索纠错、搜索推荐

数据分析：我们分析每一个商品分类下有多少个商品，select category\_id,count(\*) from products group by category\_id

（3）对海量数据进行近实时的处理

分布式：ES自动可以将海量数据分散到多台服务器上去存储和检索

海联数据的处理：分布式以后，就可以采用大量的服务器去存储和检索数据，自然而然就可以实现海量数据的处理了

近实时：检索个数据要花费1小时（这就不要近实时，离线批处理，batch-processing）；在秒级别对数据进行搜索和分析

跟分布式/海量数据相反的：lucene，单机应用，只能在单台服务器上使用，最多只能处理单台服务器可以处理的数据量

**2、Elasticsearch的适用场景**

国外

（1）维基百科，类似百度百科，牙膏，牙膏的维基百科，全文检索，高亮，搜索推荐

（2）The Guardian（国外新闻网站），类似搜狐新闻，用户行为日志（点击，浏览，收藏，评论）+社交网络数据（对某某新闻的相关看法），数据分析，给到每篇新闻文章的作者，让他知道他的文章的公众反馈（好，坏，热门，垃圾，鄙视，崇拜）

（3）Stack Overflow（国外的程序异常讨论论坛），IT问题，程序的报错，提交上去，有人会跟你讨论和回答，全文检索，搜索相关问题和答案，程序报错了，就会将报错信息粘贴到里面去，搜索有没有对应的答案

（4）GitHub（开源代码管理），搜索上千亿行代码

（5）电商网站，检索商品

（6）日志数据分析，logstash采集日志，ES进行复杂的数据分析（ELK技术，elasticsearch+logstash+kibana）

（7）商品价格监控网站，用户设定某商品的价格阈值，当低于该阈值的时候，发送通知消息给用户，比如说订阅牙膏的监控，如果高露洁牙膏的家庭套装低于50块钱，就通知我，我就去买

（8）BI系统，商业智能，Business Intelligence。比如说有个大型商场集团，BI，分析一下某某区域最近3年的用户消费金额的趋势以及用户群体的组成构成，产出相关的数张报表，\*\*区，最近3年，每年消费金额呈现100%的增长，而且用户群体85%是高级白领，开一个新商场。ES执行数据分析和挖掘，Kibana进行数据可视化

国内

（9）国内：站内搜索（电商，招聘，门户，等等），IT系统搜索（OA，CRM，ERP，等等），数据分析（ES热门的一个使用场景）

**3、Elasticsearch的特点**

（1）可以作为一个大型分布式集群（数百台服务器）技术，处理PB级数据，服务大公司；也可以运行在单机上，服务小公司

（2）Elasticsearch不是什么新技术，主要是将全文检索、数据分析以及分布式技术，合并在了一起，才形成了独一无二的ES；lucene（全文检索），商用的数据分析软件（也是有的），分布式数据库（mycat）

（3）对用户而言，是开箱即用的，非常简单，作为中小型的应用，直接3分钟部署一下ES，就可以作为生产环境的系统来使用了，数据量不大，操作不是太复杂

（4）数据库的功能面对很多领域是不够用的（事务，还有各种联机事务型的操作）；特殊的功能，比如全文检索，同义词处理，相关度排名，复杂数据分析，海量数据的近实时处理；Elasticsearch作为传统数据库的一个补充，提供了数据库所不不能提供的很多功能

## 手工画图剖析 Elasticsearch 核心概念

核心概念：NRT、索引、分片、副本等。

1、lucene和elasticsearch的前世今生

2、elasticsearch的核心概念

3、elasticsearch核心概念 vs. 数据库核心概念

**1、lucene和elasticsearch的前世今生**

lucene，最先进、功能最强大的搜索库，直接基于lucene开发，非常复杂，api复杂（实现一些简单的功能，写大量的java代码），需要深入理解原理（各种索引结构）

elasticsearch，基于lucene，隐藏复杂性，提供简单易用的restful api接口、java api接口（还有其他语言的api接口）

（1）分布式的文档存储引擎

（2）分布式的搜索引擎和分析引擎

（3）分布式，支持PB级数据

开箱即用，优秀的默认参数，不需要任何额外设置，完全开源

关于elasticsearch的一个传说，有一个程序员失业了，陪着自己老婆去英国伦敦学习厨师课程。程序员在失业期间想给老婆写一个菜谱搜索引擎，觉得lucene实在太复杂了，就开发了一个封装了lucene的开源项目，compass。后来程序员找到了工作，是做分布式的高性能项目的，觉得compass不够，就写了elasticsearch，让lucene变成分布式的系统。

**2、elasticsearch的核心概念**

（1）Near Realtime（NRT）：近实时，两个意思，从写入数据到数据可以被搜索到有一个小延迟（大概1秒）；基于es执行搜索和分析可以达到秒级



（2）Cluster：集群，包含多个节点，每个节点属于哪个集群是通过一个配置（集群名称，默认是elasticsearch）来决定的，对于中小型应用来说，刚开始一个集群就一个节点很正常

（3）Node：节点，集群中的一个节点，节点也有一个名称（默认是随机分配的），节点名称很重要（在执行运维管理操作的时候），默认节点会去加入一个名称为“elasticsearch”的集群，如果直接启动一堆节点，那么它们会自动组成一个elasticsearch集群，当然一个节点也可以组成一个elasticsearch集群

（4）Document&field：文档，es中的最小数据单元，一个document可以是一条客户数据，一条商品分类数据，一条订单数据，通常用JSON数据结构表示，每个index下的type中，都可以去存储多个document。一个document里面有多个field，每个field就是一个数据字段。

product document

{

"product\_id": "1",

"product\_name": "高露洁牙膏",

"product\_desc": "高效美白",

"category\_id": "2",

"category\_name": "日化用品"

}

（5）Index：索引，包含一堆有相似结构的文档数据，比如可以有一个客户索引，商品分类索引，订单索引，索引有一个名称。一个index包含很多document，一个index就代表了一类类似的或者相同的document。比如说建立一个product index，商品索引，里面可能就存放了所有的商品数据，所有的商品document。

（6）Type：类型，每个索引里都可以有一个或多个type，type是index中的一个逻辑数据分类，一个type下的document，都有相同的field，比如博客系统，有一个索引，可以定义用户数据type，博客数据type，评论数据type。

商品index，里面存放了所有的商品数据，商品document

但是商品分很多种类，每个种类的document的field可能不太一样，比如说电器商品，可能还包含一些诸如售后时间范围这样的特殊field；生鲜商品，还包含一些诸如生鲜保质期之类的特殊field

type，日化商品type，电器商品type，生鲜商品type

日化商品type：product\_id，product\_name，product\_desc，category\_id，category\_name

电器商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，service\_period

生鲜商品type：product\_id，product\_name，product\_desc，category\_id，category\_name，eat\_period

每一个type里面，都会包含一堆document

{

"product\_id": "2",

"product\_name": "长虹电视机",

"product\_desc": "4k高清",

"category\_id": "3",

"category\_name": "电器",

"service\_period": "1年"

}

{

"product\_id": "3",

"product\_name": "基围虾",

"product\_desc": "纯天然，冰岛产",

"category\_id": "4",

"category\_name": "生鲜",

"eat\_period": "7天"

}

（7）shard：单台机器无法存储大量数据，es可以将一个索引中的数据切分为多个shard，分布在多台服务器上存储。有了shard就可以横向扩展，存储更多数据，让搜索和分析等操作分布到多台服务器上去执行，提升吞吐量和性能。每个shard都是一个lucene index。

（8）replica：任何一个服务器随时可能故障或宕机，此时shard可能就会丢失，因此可以为每个shard创建多个replica副本。replica可以在shard故障时提供备用服务，保证数据不丢失，多个replica还可以提升搜索操作的吞吐量和性能。primary shard（建立索引时一次设置，不能修改，默认5个），replica shard（随时修改数量，默认1个），默认每个索引10个shard，5个primary shard，5个replica shard，最小的高可用配置，是2台服务器。



**3、elasticsearch核心概念 vs. 数据库核心概念**

Elasticsearch 数据库

-----------------------------------------

Document 行

Type 表

Index 库

## 在windows 上安装和启动 Elasticsearch

1、安装JDK，至少1.8.0\_73以上版本，java -version

2、下载和解压缩Elasticsearch安装包，目录结构

3、启动Elasticsearch：bin\elasticsearch.bat，es本身特点之一就是开箱即用，如果是中小型应用，数据量少，操作不是很复杂，直接启动就可以用了

4、检查ES是否启动成功：http://localhost:9200/?pretty

name: node名称

cluster\_name: 集群名称（默认的集群名称就是elasticsearch）

version.number: 5.2.0，es版本号

{

"name" : "4onsTYV",

"cluster\_name" : "elasticsearch",

"cluster\_uuid" : "nKZ9VK\_vQdSQ1J0Dx9gx1Q",

"version" : {

"number" : "5.2.0",

"build\_hash" : "24e05b9",

"build\_date" : "2017-01-24T19:52:35.800Z",

"build\_snapshot" : false,

"lucene\_version" : "6.4.0"

},

"tagline" : "You Know, for Search"

}

5、修改集群名称：elasticsearch.yml

6、下载和解压缩Kibana安装包，使用里面的开发界面，去操作elasticsearch，作为我们学习es知识点的一个主要的界面入口

7、启动Kibana：bin\kibana.bat

8、进入Dev Tools界面

9、GET \_cluster/health

## 快速入门-电商案例

### 快速入门-电商案例一

电商的商品管理：集群健康检查以及文档的 CRUD

1、document数据格式

2、电商网站商品管理案例：背景介绍

3、简单的集群管理

4、商品的CRUD操作（document CRUD操作）

**1、document数据格式**

面向文档的搜索分析引擎

（1）应用系统的数据结构都是面向对象的，复杂的

（2）对象数据存储到数据库中，只能拆解开来，变为扁平的多张表，每次查询的时候还得还原回对象格式，相当麻烦

（3）ES是面向文档的，文档中存储的数据结构，与面向对象的数据结构是一样的，基于这种文档数据结构，es可以提供复杂的索引，全文检索，分析聚合等功能

（4）es的document用json数据格式来表达

public class Employee {

private String email;

private String firstName;

private String lastName;

private EmployeeInfo info;

private Date joinDate;

}

private class EmployeeInfo {

private String bio; // 性格

private Integer age;

private String[] interests; // 兴趣爱好

}

EmployeeInfo info = new EmployeeInfo();

info.setBio("curious and modest");

info.setAge(30);

info.setInterests(new String[]{"bike", "climb"});

Employee employee = new Employee();

employee.setEmail("zhangsan@sina.com");

employee.setFirstName("san");

employee.setLastName("zhang");

employee.setInfo(info);

employee.setJoinDate(new Date());

employee对象：里面包含了Employee类自己的属性，还有一个EmployeeInfo对象

两张表：employee表，employee\_info表，将employee对象的数据重新拆开来，变成Employee数据和EmployeeInfo数据

employee表：email，first\_name，last\_name，join\_date，4个字段

employee\_info表：bio，age，interests，3个字段；此外还有一个外键字段，比如employee\_id，关联着employee表

{

"email": "zhangsan@sina.com",

"first\_name": "san",

"last\_name": "zhang",

"info": {

"bio": "curious and modest",

"age": 30,

"interests": [ "bike", "climb" ]

},

"join\_date": "2017/01/01"

}

我们就明白了es的document数据格式和数据库的关系型数据格式的区别

**2、电商网站商品管理案例背景介绍**

有一个电商网站，需要为其基于ES构建一个后台系统，提供以下功能：

（1）对商品信息进行CRUD（增删改查）操作

（2）执行简单的结构化查询

（3）可以执行简单的全文检索，以及复杂的phrase（短语）检索

（4）对于全文检索的结果，可以进行高亮显示

（5）对数据进行简单的聚合分析

**3、简单的集群管理**

（1）快速检查集群的健康状况

es提供了一套api，叫做cat api，可以查看es中各种各样的数据

GET /\_cat/health?v

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488006741 15:12:21 elasticsearch yellow 1 1 1 1 0 0 1 0 - 50.0%

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488007113 15:18:33 elasticsearch green 2 2 2 1 0 0 0 0 - 100.0%

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1488007216 15:20:16 elasticsearch yellow 1 1 1 1 0 0 1 0 - 50.0%

如何快速了解集群的健康状况？green、yellow、red？

1. green：每个索引的primary shard和replica shard都是active状态的

2. yellow：每个索引的primary shard都是active状态的，但是部分replica shard不是active状态，处于不可用的状态

3. red：不是所有索引的primary shard都是active状态的，部分索引有数据丢失了

为什么现在会处于一个yellow状态？

我们现在就一个笔记本电脑，就启动了一个es进程，相当于就只有一个node。现在es中有一个index，就是kibana自己内置建立的index。由于默认的配置是给每个index分配5个primary shard和5个replica shard，而且primary shard和replica shard不能在同一台机器上（为了容错）。现在kibana自己建立的index是1个primary shard和1个replica shard。当前就一个node，所以只有1个primary shard被分配了和启动了，但是一个replica shard没有第二台机器去启动。

做一个小实验：此时只要启动第二个es进程，就会在es集群中有2个node，然后那1个replica shard就会自动分配过去，然后cluster status就会变成green状态。

（2）快速查看集群中有哪些索引

GET /\_cat/indices?v

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

（3）简单的索引操作

创建索引：PUT /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open test\_index XmS9DTAtSkSZSwWhhGEKkQ 5 1 0 0 650b 650b

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

删除索引：DELETE /test\_index?pretty

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open .kibana rUm9n9wMRQCCrRDEhqneBg 1 1 1 0 3.1kb 3.1kb

**4、商品的CRUD操作**

PS：目前这里还没有

（1）新增商品：新增文档，建立索引

语法：

PUT /index/type/id

{

"json数据"

}

eg：

PUT /ecommerce/product/1

{

"name" : "gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

"tags": [ "meibai", "fangzhu" ]

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

// 写一个 primary shard，然后 replica shard 也需要有这个数据。一个节点所以 succesful 是 1， 两个 shard 所以\_shards的total值是 2

PUT /ecommerce/product/2

{

"name" : "jiajieshi yagao",

"desc" : "youxiao fangzhu",

"price" : 25,

"producer" : "jiajieshi producer",

"tags": [ "fangzhu" ]

}

PUT /ecommerce/product/3

{

"name" : "zhonghua yagao",

"desc" : "caoben zhiwu",

"price" : 40,

"producer" : "zhonghua producer",

"tags": [ "qingxin" ]

}

es会自动建立index和type，不需要提前创建，而且es默认会对document每个field都建立倒排索引，让其可以被搜索

（2）查询商品：检索文档

语法：GET /{index}/{type}/{id}

GET /ecommerce/product/1

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"name": "gaolujie yagao",

"desc": "gaoxiao meibai",

"price": 30,

"producer": "gaolujie producer",

"tags": [

"meibai",

"fangzhu"

]

}

}

（3）修改商品：替换文档

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao",

"desc" : "gaoxiao meibai",

"price" : 30,

"producer" : "gaolujie producer",

"tags": [ "meibai", "fangzhu" ]

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

PUT /ecommerce/product/1

{

"name" : "jiaqiangban gaolujie yagao"

}

替换方式有一个不好，即使必须带上所有的field，才能去进行信息的修改

（4）修改商品：更新文档

这里是进行 Partial Update 操作：

POST /ecommerce/product/1/\_update

{

"doc": {

"name": "jiaqiangban gaolujie yagao"

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 8,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

我的风格，其实有选择的情况下，不太喜欢念ppt，或者照着文档做，或者直接粘贴写好的代码，尽量是纯手敲代码

（5）删除商品：删除文档

DELETE /ecommerce/product/1

{

"found": true,

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"\_version": 9,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

{

"\_index": "ecommerce",

"\_type": "product",

"\_id": "1",

"found": false

}

### 快速入门-电商案例二

主要是对查询的一些语法的体验

1、query string search

2、query DSL

3、query filter

4、full-text search

5、phrase search

6、highlight search

题外话：把英文翻译成中文，让我觉得很别扭，term翻译成词项

**1、query string search**

搜索全部商品：

GET /ecommerce/product/\_search

took：耗费了几毫秒

timed\_out：是否超时，这里是没有

\_shards：数据拆成了5个分片，所以对于搜索请求，会打到所有的primary shard（或者是它的某个replica shard也可以）

hits.total：查询结果的数量，3个document

hits.max\_score：score的含义，就是document对于一个search的相关度的匹配分数，越相关，就越匹配，分数也高

hits.hits：包含了匹配搜索的document的详细数据

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1,  "hits": [  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "2",  "\_score": 1,  "\_source": {  "name": "jiajieshi yagao",  "desc": "youxiao fangzhu",  "price": 25,  "producer": "jiajieshi producer",  "tags": [  "fangzhu"  ]  }  },  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "1",  "\_score": 1,  "\_source": {  "name": "gaolujie yagao",  "desc": "gaoxiao meibai",  "price": 30,  "producer": "gaolujie producer",  "tags": [  "meibai",  "fangzhu"  ]  }  },  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "3",  "\_score": 1,  "\_source": {  "name": "zhonghua yagao",  "desc": "caoben zhiwu",  "price": 40,  "producer": "zhonghua producer",  "tags": [  "qingxin"  ]  }  }  ]  }  } |

query string search的由来，因为search参数都是以http请求的query string来附带的

搜索商品名称中包含yagao的商品，而且按照售价降序排序：GET /ecommerce/product/\_search?q=name:yagao&sort=price:desc

适用于临时的在命令行使用一些工具，比如curl，快速的发出请求，来检索想要的信息；但是如果查询请求很复杂，是很难去构建的

在生产环境中，几乎很少使用query string search

**2、query DSL**

DSL：Domain Specified Language，特定领域的语言

http request body：请求体，可以用json的格式来构建查询语法，比较方便，可以构建各种复杂的语法，比query string search肯定强大多了

查询所有的商品

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} }

}

查询名称包含yagao的商品，同时按照价格降序排序

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"name" : "yagao"

}

},

"sort": [

{ "price": "desc" }

]

}

分页查询商品，总共3条商品，假设每页就显示1条商品，现在显示第2页，所以就查出来第2个商品

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} },

"from": 1,

"size": 1

}

指定要查询出来商品的名称和价格就可以

GET /ecommerce/product/\_search

{

"query": { "match\_all": {} },

"\_source": ["name", "price"]

}

更加适合生产环境的使用，可以构建复杂的查询

**3、query filter**

搜索商品名称包含yagao，而且售价大于25元的商品

GET /ecommerce/product/\_search

{

"query" : {

"bool" : {

"must" : {

"match" : {

"name" : "yagao"

}

},

"filter" : {

"range" : {

"price" : { "gt" : 25 }

}

}

}

}

}

**4、full-text search（全文检索）**

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "yagao producer"

}

}

}

PS：尽量，无论是学什么技术，比如说你当初学java，学linux，学shell，学javascript，学hadoop。。。。一定自己动手，特别是手工敲各种命令和代码，切记切记，减少复制粘贴的操作。只有自己动手手工敲，学习效果才最好。

**过程：**producer这个字段，会先被拆解，建立倒排索引如下所示

special 4

yagao 4

producer 1,2,3,4

gaolujie 1

zhognhua 3

jiajieshi 2

搜索的单词yagao producer 会被拆解为 yagao和producer

结果：

|  |
| --- |
| {  "took": 4,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 4,  "max\_score": 0.70293105,  "hits": [  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "4",  "\_score": 0.70293105,  "\_source": {  "name": "special yagao",  "desc": "special meibai",  "price": 50,  "producer": "special yagao producer",  "tags": [  "meibai"  ]  }  },  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "1",  "\_score": 0.25811607,  "\_source": {  "name": "gaolujie yagao",  "desc": "gaoxiao meibai",  "price": 30,  "producer": "gaolujie producer",  "tags": [  "meibai",  "fangzhu"  ]  }  },  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "3",  "\_score": 0.25811607,  "\_source": {  "name": "zhonghua yagao",  "desc": "caoben zhiwu",  "price": 40,  "producer": "zhonghua producer",  "tags": [  "qingxin"  ]  }  },  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "2",  "\_score": 0.1805489,  "\_source": {  "name": "jiajieshi yagao",  "desc": "youxiao fangzhu",  "price": 25,  "producer": "jiajieshi producer",  "tags": [  "fangzhu"  ]  }  }  ]  }  } |

**5、phrase search（短语搜索）**

跟全文检索相对应，相反，全文检索会将输入的搜索串拆解开来，去倒排索引里面去一一匹配，只要能匹配上任意一个拆解后的单词，就可以作为结果返回

phrase search，要求输入的搜索串，必须在指定的字段文本中，完全包含一模一样的，才可以算匹配，才能作为结果返回

GET /ecommerce/product/\_search

{

"query" : {

"match\_phrase" : {

"producer" : "yagao producer"

}

}

}

结果：

|  |
| --- |
| {  "took": 11,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 0.70293105,  "hits": [  {  "\_index": "ecommerce",  "\_type": "product",  "\_id": "4",  "\_score": 0.70293105,  "\_source": {  "name": "special yagao",  "desc": "special meibai",  "price": 50,  "producer": "special yagao producer",  "tags": [  "meibai"  ]  }  }  ]  }  } |

**6、highlight search（高亮搜索结果）**

GET /ecommerce/product/\_search

{

"query" : {

"match" : {

"producer" : "producer"

}

},

"highlight": {

"fields" : {

"producer" : {}

}

}

}

### 快速入门-电商案例三

主要是体验对分析方面的工程

**第一个分析需求：**

计算每个tag下的商品数量

GET /ecommerce/product/\_search

{

"aggs": {

"group\_by\_tags": {

"terms": { "field": "tags" }

}

}

}

上面执行后就会报异常。

fileddata的默认值是禁止的，如果需要聚合，就需要设置为 true，

因此将文本field的fielddata属性设置为true

PUT /ecommerce/\_mapping/product

{

"properties": {

"tags": {

"type": "text",

"fielddata": true

}

}

}

GET /ecommerce/product/\_search

{

"size": 0,

"aggs": {

"all\_tags": {

"terms": { "field": "tags" }

}

}

}

结果：

|  |
| --- |
| {  "took": 20,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 4,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_tags": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "fangzhu",  "doc\_count": 2  },  {  "key": "meibai",  "doc\_count": 2  },  {  "key": "qingxin",  "doc\_count": 1  }  ]  }  }  } |

**第二个聚合分析的需求：**

对名称中包含yagao的商品，计算每个tag下的商品数量

GET /ecommerce/product/\_search

{

"size": 0,

"query": {

"match": {

"name": "yagao"

}

},

"aggs": {

"all\_tags": {

"terms": {

"field": "tags"

}

}

}

}

**第三个聚合分析的需求：**

先分组，再算每组的平均值，计算每个tag下的商品的平均价格

GET /ecommerce/product/\_search

{

"size": 0,

"aggs" : {

"group\_by\_tags" : {

"terms" : { "field" : "tags" },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

|  |
| --- |
| {  "took": 8,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 4,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_tags": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "fangzhu",  "doc\_count": 2,  "avg\_price": {  "value": 27.5  }  },  {  "key": "meibai",  "doc\_count": 2,  "avg\_price": {  "value": 40  }  },  {  "key": "qingxin",  "doc\_count": 1,  "avg\_price": {  "value": 40  }  }  ]  }  }  } |

**第四个数据分析需求：**

计算每个tag下的商品的平均价格，并且按照平均价格降序排序

GET /ecommerce/product/\_search

{

"size": 0,

"aggs" : {

"all\_tags" : {

"terms" : { "field" : "tags", "order": { "avg\_price": "desc" } },

"aggs" : {

"avg\_price" : {

"avg" : { "field" : "price" }

}

}

}

}

}

我们现在全部都是用es的restful api在学习和讲解es的所欲知识点和功能点，但是没有使用一些编程语言去讲解（比如java），原因有以下：

1、es最重要的api，让我们进行各种尝试、学习甚至在某些环境下进行使用的api，就是restful api。如果你学习不用es restful api，比如我上来就用java api来讲es，也是可以的，但是你根本就漏掉了es知识的一大块，你都不知道它最重要的restful api是怎么用的

2、讲知识点，用es restful api，更加方便，快捷，不用每次都写大量的java代码，能加快讲课的效率和速度，更加易于同学们关注es本身的知识和功能的学习

3、我们通常会讲完es知识点后，开始详细讲解java api，如何用java api执行各种操作

4、我们每个篇章都会搭配一个项目实战，项目实战是完全基于java去开发的真实项目和系统

**第五个数据分析需求：**

按照指定的价格范围区间进行分组，然后在每组内再按照tag进行分组，最后再计算每组的平均价格

GET /ecommerce/product/\_search

{

"size": 0,

"aggs": {

"group\_by\_price": {

"range": {

"field": "price",

"ranges": [

{

"from": 0,

"to": 20

},

{

"from": 20,

"to": 40

},

{

"from": 40,

"to": 50

}

]

},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags"

},

"aggs": {

"average\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

## Elasticsearch 的基础分布式架构

1、Elasticsearch对复杂分布式机制的透明隐藏特性

2、Elasticsearch的垂直扩容与水平扩容

3、增减或减少节点时的数据rebalance

4、master节点

5、节点对等的分布式架构



**1、Elasticsearch对复杂分布式机制的透明隐藏特性**

Elasticsearch是一套分布式的系统，分布式是为了应对大数据量

隐藏了复杂的分布式机制

首先：分片机制（我们之前随随便便就将一些document插入到es集群中去了，我们有没有care过数据怎么进行分片的，数据到哪个shard中去）

cluster discovery（集群发现机制，我们之前在做那个集群status从yellow转green的实验里，直接启动了第二个es进程，那个进程作为一个node自动就发现了集群，并且加入了进去，还接受了部分数据，replica shard）

shard负载均衡（举例，假设现在有3个节点，总共有25个shard要分配到3个节点上去，es会自动进行均匀分配，以保持每个节点的均衡的读写负载请求）

shard副本，请求路由，集群扩容，shard重分配

**2、Elasticsearch的垂直扩容与水平扩容**

垂直扩容：采购更强大的服务器，成本非常高昂，而且会有瓶颈，假设世界上最强大的服务器容量就是10T，但是当你的总数据量达到5000T的时候，你要采购多少台最强大的服务器啊

**水平扩容**：业界经常采用的方案，采购越来越多的普通服务器，性能比较一般，但是很多普通服务器组织在一起，就能构成强大的计算和存储能力

普通服务器：1T，1万，100万

强大服务器：10T，50万，500万

扩容对应用程序的透明性

**3、增减或减少节点时的数据rebalance**

保持负载均衡

**4、master节点**

（1）创建或删除索引

（2）增加或删除节点

**5、节点平等的分布式架构**

（1）节点对等，每个节点都能接收所有的请求

（2）自动请求路由

（3）响应收集

## shard&replica 机制再次梳理以及单 node 环境中创建 index 图解

1、shard&replica机制再次梳理

2、图解单node环境下创建index是什么样子的

**1、shard&replica机制再次梳理**

（1）index包含多个shard

（2）每个shard都是一个最小工作单元，承载部分数据，lucene实例，完整的建立索引和处理请求的能力，一个 Document 只会在一个 shard 里面。

（3）增减节点时，shard会自动在nodes中负载均衡

（4）primary shard和replica shard，每个document肯定只存在于某一个primary shard以及其对应的replica shard中，不可能存在于多个primary shard

（5）replica shard是primary shard的副本，负责容错，以及承担读请求负载

（6）primary shard的数量在创建索引的时候就固定了，replica shard的数量可以随时修改

（7）primary shard的默认数量是5，replica默认是1，默认有10个shard，5个primary shard，5个replica shard

（8）primary shard 不能和自己的replica shard放在同一个节点上（否则节点宕机，primary shard和副本都丢失，起不到容错的作用），但是可以和其他primary shard的replica shard放在同一个节点上

**2、图解单node环境下创建index是什么样子的**

（1）单node环境下，创建一个index，有3个primary shard，3个replica shard

（2）集群status是yellow

（3）这个时候，只会将3个primary shard分配到仅有的一个node上去，另外3个replica shard是无法分配的

（4）集群可以正常工作，但是一旦出现节点宕机，数据全部丢失，而且集群不可用，无法承接任何请求

PUT /test\_index

{

"settings" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 1

}

}



## 图解 2 个 node 环境下 replica shard 是如何分配的

1、图解2个node环境下replica shard是如何分配的

其实重点就是，如果是多个服务器的话，replica shard 就会被分配。根据 primary shard 和 replica shard 不能再一台服务器上的原则。

（1）replica shard分配：3个primary shard，3个replica shard，1 node

（2）primary ---> replica同步

（3）读请求：primary/replica



## 图解横向扩容过程，如何超出扩容极限，以及如何提升容错性

（1）primary&replica自动负载均衡，6个shard，3 primary，3 replica

（2）每个node有更少的shard，IO/CPU/Memory资源给每个shard分配更多，每个shard性能更好

（3）扩容的极限，6个shard（3 primary，3 replica），最多扩容到6台机器，每个shard可以占用单台服务器的所有资源，性能最好

（4）超出扩容极限，动态修改replica数量，9个shard（3primary，6 replica），扩容到9台机器，比3台机器时，拥有3倍的读吞吐量

（5）3台机器下，9个shard（3 primary，6 replica），资源更少，但是容错性更好，最多容纳2台机器宕机，6个shard只能容纳0台机器宕机

（6）这里的这些知识点，你综合起来看，就是说，一方面告诉你扩容的原理，怎么扩容，怎么提升系统整体吞吐量；另一方面要考虑到系统的容错性，怎么保证提高容错性，让尽可能多的服务器宕机，保证数据不丢失



## 图解 Elasticsearch 容错机制：master 选举，replica 容错，数据恢复

（1）9 shard，3 node

（2）master node宕机，自动master选举，red

（3）replica容错：新master将replica提升为primary shard，yellow

（4）重启宕机node，master copy replica到该node，使用原有的shard并同步宕机后的修改，green

## 初步解析 document 的核心元数据以及图解剖析 index 创建反例

1、\_index元数据

2、\_type元数据

3、\_id元数据

**1、\_index元数据**

（1）代表一个document存放在哪个index中

（2）类似的数据放在一个索引，非类似的数据放不同索引：product index（包含了所有的商品），sales index（包含了所有的商品销售数据），inventory index（包含了所有库存相关的数据）。如果你把比如product，sales，human resource（employee），全都放在一个大的index里面，比如说company index，不合适的。

（3）index中包含了很多类似的document：类似是什么意思，其实指的就是说，这些document的fields很大一部分是相同的，你说你放了3个document，每个document的fields都完全不一样，这就不是类似了，就不太适合放到一个index里面去了。

（4）索引名称必须是小写的，不能用下划线开头，不能包含逗号：product，website，blog

**2、\_type元数据**

（1）代表document属于index中的哪个类别（type）

（2）一个索引通常会划分为多个type，逻辑上对index中有些许不同的几类数据进行分类：因为一批相同的数据，可能有很多相同的fields，但是还是可能会有一些轻微的不同，可能会有少数fields是不一样的，举个例子，就比如说，商品，可能划分为电子商品，生鲜商品，日化商品，等等。

（3）type名称可以是大写或者小写，但是同时不能用下划线开头，不能包含逗号

**3、\_id元数据**

（1）代表document的唯一标识，与index和type一起，可以唯一标识和定位一个document

（2）我们可以手动指定document的id（put /index/type/id），也可以不指定，由es自动为我们创建一个id

## document id 的手动执行与自动生成两种方式解析

1、手动指定document id

2、自动生成document id

**1、手动指定document id**

（1）根据应用情况来说，是否满足手动指定document id的前提：

一般来说，是从某些其他的系统中，导入一些数据到es时，会采取这种方式，就是使用系统中已有数据的唯一标识，作为es中document的id。举个例子，比如说，我们现在在开发一个电商网站，做搜索功能，或者是OA系统，做员工检索功能。这个时候，数据首先会在网站系统或者IT系统内部的数据库中，会先有一份，此时就肯定会有一个数据库的primary key（自增长，UUID，或者是业务编号）。如果将数据导入到es中，此时就比较适合采用数据在数据库中已有的primary key。

如果说，我们是在做一个系统，这个系统主要的数据存储就是es一种，也就是说，数据产生出来以后，可能就没有id，直接就放es一个存储，那么这个时候，可能就不太适合说手动指定document id的形式了，因为你也不知道id应该是什么，此时可以采取下面要讲解的让es自动生成id的方式。

（2）put /index/type/id

PUT /test\_index/test\_type/2

{

"test\_content": "my test"

}

**2、自动生成document id**

（1）post /index/type

POST /test\_index/test\_type

{

"test\_content": "my test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "AVp4RN0bhjxldOOnBxaE",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

（2）自动生成的id，长度为20个字符，URL安全，base64编码，GUID，分布式系统并行生成时不可能会发生冲突

## document 的\_source元数据以及定制返回结果解析

**1、\_source元数据**

put /test\_index/test\_type/1

{

"test\_field1": "test field1",

"test\_field2": "test field2"

}

get /test\_index/test\_type/1

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field1": "test field1",

"test\_field2": "test field2"

}

}

\_source元数据：就是说，我们在创建一个document的时候，使用的那个放在request body中的json串，默认情况下，在get的时候，会原封不动的给我们返回回来。

------------------------------------------------------------------------------------------------------------------

**2、定制返回结果**

定制返回的结果，指定\_source中，返回哪些field

GET /test\_index/test\_type/1?\_source=test\_field1,test\_field2

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "1",

"\_version": 2,

"found": true,

"\_source": {

"test\_field2": "test field2"

}

}

## document 的全量替换、强制创建以及图解 lazy delete 机制

1、document的全量替换

2、document的强制创建

3、document的删除

**1、document的全量替换**

（1）语法与创建文档是一样的，如果document id不存在，那么就是创建；如果document id已经存在，那么就是全量替换操作，替换document的json串内容

（2）document是不可变的，如果要修改document的内容，第一种方式就是全量替换，直接对document重新建立索引，替换里面所有的内容

（3）es会将老的document标记为deleted，然后新增我们给定的一个document，当我们创建越来越多的document的时候，es会在适当的时机在后台自动删除标记为deleted的document

**2、document的强制创建**

（1）创建文档与全量替换的语法是一样的，有时我们只是想新建文档，不想替换文档，如果强制进行创建呢？

（2）PUT /index/type/id?op\_type=create，PUT /index/type/id/\_create

**3、document的删除**

（1）DELETE /index/type/id

（2）不会理解物理删除，只会将其标记为deleted，当数据越来越多的时候，在后台自动删除



## ES 并发冲突-乐观锁

### 深度剖析 Elasticsearch 并发冲突问题



### 深度图解剖析悲观锁与乐观锁两种并发控制方案

### 图解 Elasticsearch 内部如何基于\_version 进行乐观锁并发控制

（1）\_version元数据

PUT /test\_index/test\_type/6

{

"test\_field": "test test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

第一次创建一个document的时候，它的\_version内部版本号就是1；以后，每次对这个document执行修改或者删除操作，都会对这个\_version版本号自动加1；哪怕是删除，也会对这条数据的版本号加1

{

"found": true,

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "6",

"\_version": 4,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

我们会发现，在删除一个document之后，可以从一个侧面证明，它不是立即物理删除掉的，因为它的一些版本号等信息还是保留着的。先删除一条document，再重新创建这条document，其实会在delete version基础之上，再把version号加1



### 上机动手实战基于\_version 进行乐观锁并发控制

1、上机动手实战演练基于\_version进行乐观锁并发控制

（1）先构造一条数据出来

PUT /test\_index/test\_type/7

{

"test\_field": "test test"

}

（2）模拟两个客户端，都获取到了同一条数据

GET test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test test"

}

}

（3）其中一个客户端，先更新了一下这个数据

同时带上数据的版本号，确保说，es中的数据的版本号，跟客户端中的数据的版本号是相同的，才能修改

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

（4）另外一个客户端，尝试基于version=1的数据去进行修改，同样带上version版本号，进行乐观锁的并发控制

PUT /test\_index/test\_type/7?version=1

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][7]: version conflict, current version [2] is different than the one provided [1]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "3",

"index": "test\_index"

},

"status": 409

}

（5）在乐观锁成功阻止并发问题之后，尝试正确的完成更新

GET /test\_index/test\_type/7

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

基于最新的数据和版本号，去进行修改，修改后，带上最新的版本号，可能这个步骤会需要反复执行好几次，才能成功，特别是在多线程并发更新同一条数据很频繁的情况下

PUT /test\_index/test\_type/7?version=2

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "7",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

### 上机动手实战基于 external version 进行乐观锁并发控制

1、上机动手实战演练基于external version进行乐观锁并发控制

external version

es提供了一个feature，就是说，你可以不用它提供的内部\_version版本号来进行并发控制，可以基于你自己维护的一个版本号来进行并发控制。举个列子，加入你的数据在mysql里也有一份，然后你的应用系统本身就维护了一个版本号，无论是什么自己生成的，程序控制的。这个时候，你进行乐观锁并发控制的时候，可能并不是想要用es内部的\_version来进行控制，而是用你自己维护的那个version来进行控制。

?version=1

?version=1&version\_type=external

version\_type=external，唯一的区别在于，\_version，只有当你提供的version与es中的\_version一模一样的时候，才可以进行修改，只要不一样，就报错；当version\_type=external的时候，只有当你提供的version比es中的\_version大的时候，才能完成修改

es，\_version=1，?version=1，才能更新成功

es，\_version=1，?version>1&version\_type=external，才能成功，比如说?version=2&version\_type=external

（1）先构造一条数据

PUT /test\_index/test\_type/8

{

"test\_field": "test"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

（2）模拟两个客户端同时查询到这条数据

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 1,

"found": true,

"\_source": {

"test\_field": "test"

}

}

（3）第一个客户端先进行修改，此时客户端程序是在自己的数据库中获取到了这条数据的最新版本号，比如说是2

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 1"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

（4）模拟第二个客户端，同时拿到了自己数据库中维护的那个版本号，也是2，同时基于version=2发起了修改

PUT /test\_index/test\_type/8?version=2&version\_type=external

{

"test\_field": "test client 2"

}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[test\_type][8]: version conflict, current version [2] is higher or equal to the one provided [2]",

"index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",

"shard": "1",

"index": "test\_index"

},

"status": 409

}

（5）在并发控制成功后，重新基于最新的版本号发起更新

GET /test\_index/test\_type/8

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 2,

"found": true,

"\_source": {

"test\_field": "test client 1"

}

}

PUT /test\_index/test\_type/8?version=3&version\_type=external

{

"test\_field": "test client 2"

}

{

"\_index": "test\_index",

"\_type": "test\_type",

"\_id": "8",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": false

}

## ES 的 Partial Update 操作

### 图解 partial update 实现原理以及动手实战演练

1、什么是partial update？

PUT /index/type/id，创建文档&替换文档，就是一样的语法

一般对应到应用程序中，每次的执行流程基本是这样的：

（1）应用程序先发起一个get请求，获取到document，展示到前台界面，供用户查看和修改

（2）用户在前台界面修改数据，发送到后台

（3）后台代码，会将用户修改的数据在内存中进行执行，然后封装好修改后的全量数据

（4）然后发送PUT请求，到es中，进行全量替换

（5）es将老的document标记为deleted，然后重新创建一个新的document

那么partial update是什么了。下面是它的语法格式：

POST /{index}/{type}/\_id/\_update

{

"doc": {

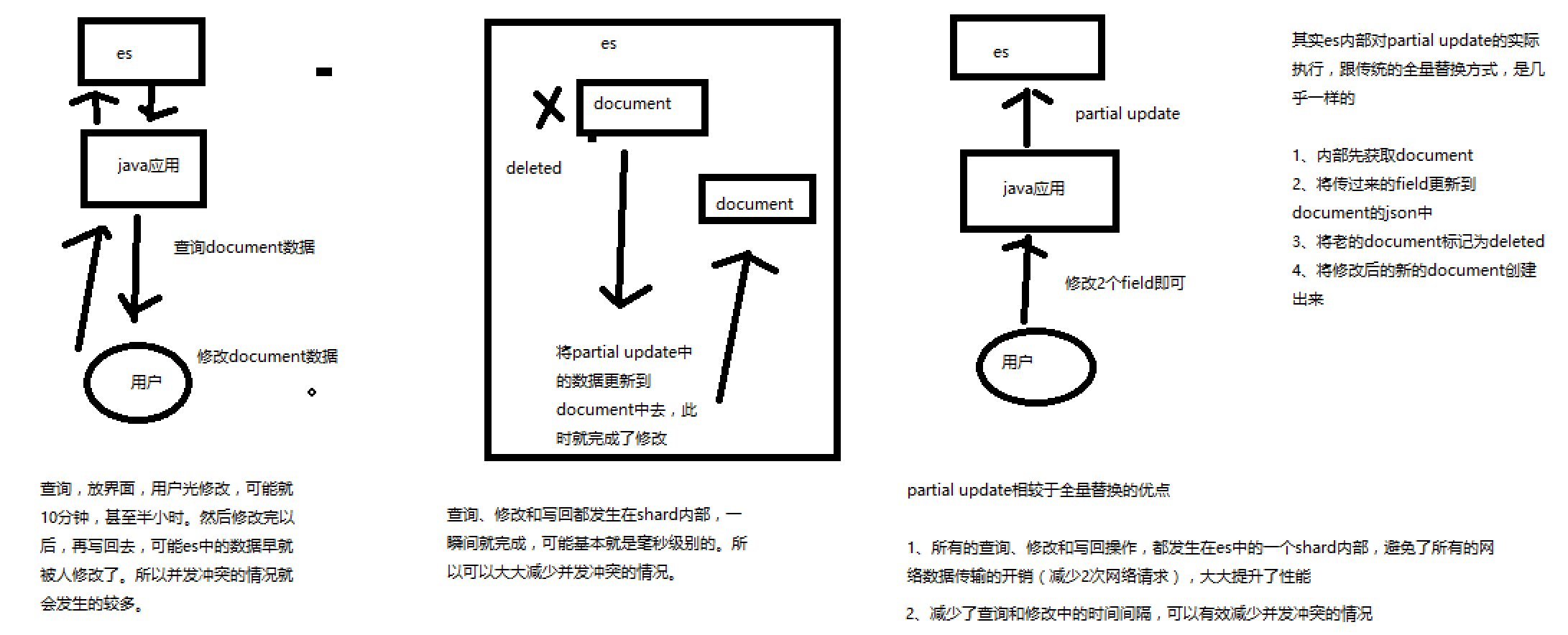
"要修改的少数几个field即可，不需要全量的数据"

}

}

看起来，好像就比较方便了，每次就传递少数几个发生修改的field即可，不需要将全量的document数据发送过去

2、图解partial update实现原理以及其优点

partial update，看起来很方便的操作，实际内部的原理是什么样子的，然后它的优点是什么

最左边画的是一般的应用程序进行修改查询的流程。

**优点**：1. 所有的查询、修改和写回操作，都在一个 es 内部的一个 shard 内执行完毕。

2. 减少了查询和修改中的时间间隔。

3、上机动手实战演练partial update

先添加一条数据

PUT /test\_index/test\_type/10

{

"test\_field1": "test1",

"test\_field2": "test2"

}

然后进行 partitial update 操作：

POST /test\_index/test\_type/10/\_update

{

"doc": {

"test\_field2": "updated test2"

}

}

### 上机动手实战演练基于 groovy 脚本执行 partial update

es，其实是有个内置的脚本支持的，可以基于groovy脚本实现各种各样的复杂操作

基于groovy脚本，如何执行partial update

**es scripting module**，我们会在高手进阶篇去讲解，这里就只是初步讲解一下

先增加一条数据

PUT /test\_index/test\_type/11

{

"num": 0,

"tags": []

}

（1）内置脚本

POST /test\_index/test\_type/11/\_update

{

"script" : "ctx.\_source.num+=1"

}

结果：

|  |
| --- |
| {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "11",  "\_version": 2,  "found": true,  "\_source": {  "num": 1,  "tags": []  }  } |

（2）外部脚本

STEP1: 在 ES\_HOME目录下的 config/scripts 目录下新建 test-add-tags.groovy。内容如下：

ctx.\_source.tags+=new\_tag

STEP2: 然后执行如下命令

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-add-tags",

"params": {

"new\_tag": "tag1"

}

}

}

（3）用脚本删除文档

STEP1: 在 ES\_HOME目录下的 config/scripts 目录下新建 test-delete-document.groovy。内容如下：

ctx.op = ctx.\_source.num == count ? 'delete' : 'none'

STEP2: 然后执行如下命令

POST /test\_index/test\_type/11/\_update

{

"script": {

"lang": "groovy",

"file": "test-delete-document",

"params": {

"count": 1

}

}

}

（4）upsert操作

进行 partital update 操作，会出这个文档不存在的错误。

POST /test\_index/test\_type/11/\_update

{

"doc": {

"num": 1

}

}

Result:

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "document\_missing\_exception",  "reason": "[test\_type][11]: document missing",  "index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",  "shard": "4",  "index": "test\_index"  }  ],  "type": "document\_missing\_exception",  "reason": "[test\_type][11]: document missing",  "index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",  "shard": "4",  "index": "test\_index"  },  "status": 404  } |

如果指定的document不存在，就执行upsert中的初始化操作；如果指定的document存在，就执行doc或者script指定的partial update操作

执行 upsert 操作如下命令格式：

POST /test\_index/test\_type/11/\_update

{

"script" : "ctx.\_source.num+=1",

"upsert": {

"num": 0,

"tags": []

}

}

### 图解 partial update 乐观锁并发控制原理以及相关操作讲解

（1）partial update内置乐观锁并发控制

partial update也会自动执行乐观锁的操作来保证并发情况的数据的准确

（2）retry\_on\_conflict

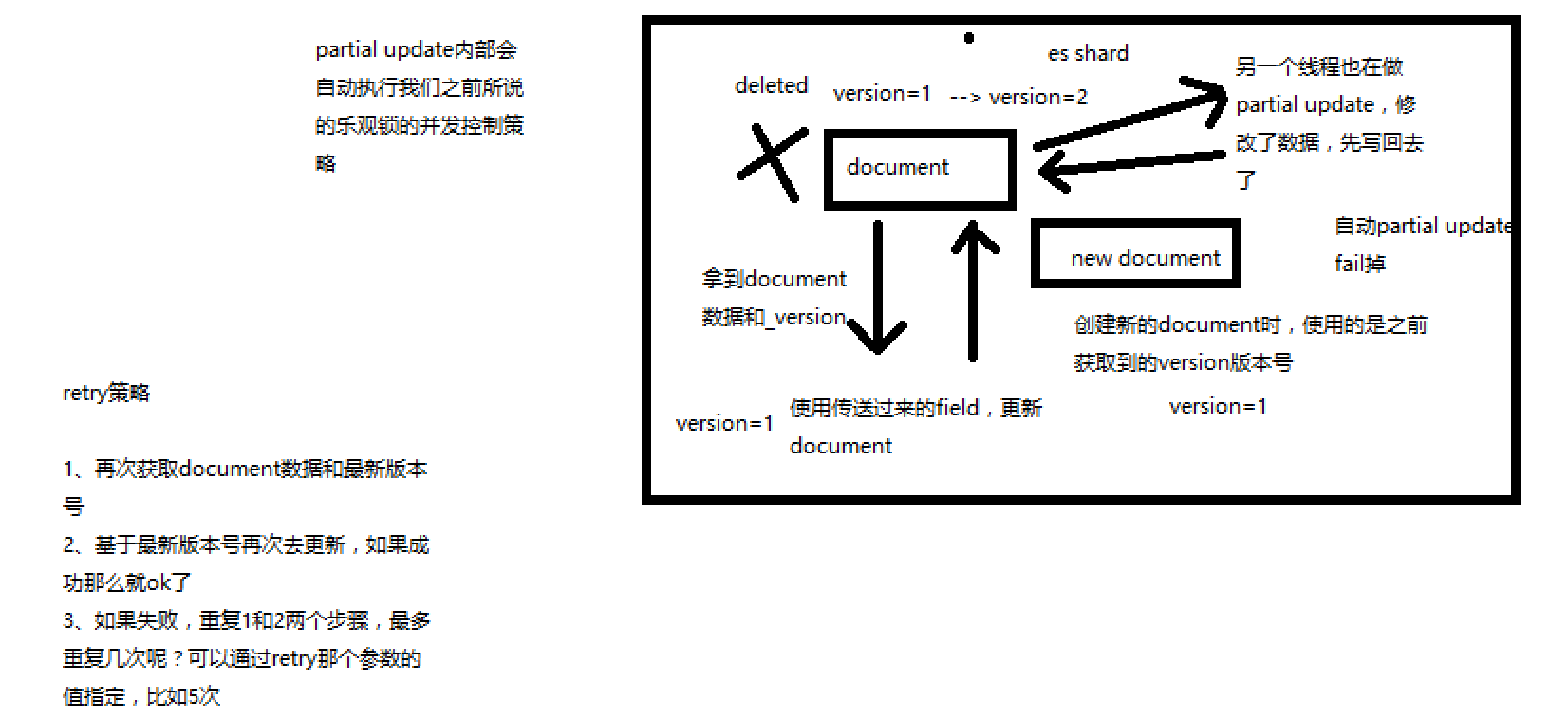
STEP1：再次获取Document 数据和最新版本号

STEP2：基于最新版本号再次更新，如果成功就 OK。

STEP3：如果失败就重复 1 和 2 步骤，知道重复到retry\_on\_conflict的值

（3）\_version

post /index/type/id/\_update?retry\_on\_conflict=5&version=6



## 上级动手实战演练 mget 批量查询 API

1、批量查询的好处

就是一条一条的查询，比如说要查询100条数据，那么就要发送100次网络请求，这个开销还是很大的

如果进行批量查询的话，查询100条数据，就只要发送1次网络请求，网络请求的性能开销缩减100倍

2、mget的语法

（1）一条一条的查询

GET /test\_index/test\_type/1

GET /test\_index/test\_type/2

（2）mget批量查询

GET /\_mget

{

"docs" : [

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_index" : "test\_index",

"\_type" : "test\_type",

"\_id" : 2

}

]

}

Result：

|  |
| --- |
| {  "docs": [  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "1",  "\_version": 2,  "found": true,  "\_source": {  "test\_field1": "test field1",  "test\_field2": "test field2"  }  },  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "2",  "\_version": 1,  "found": true,  "\_source": {  "test\_content": "my test"  }  }  ]  } |

（3）如果查询的document是一个index下的不同type种的话

GET /test\_index/\_mget

{

"docs" : [

{

"\_type" : "test\_type",

"\_id" : 1

},

{

"\_type" : "test\_type",

"\_id" : 2

}

]

}

（4）如果查询的数据都在同一个index下的同一个type下，最简单了

GET /test\_index/test\_type/\_mget

{

"ids": [1, 2]

}

3、mget的重要性

可以说mget是很重要的，一般来说，在进行查询的时候，如果一次性要查询多条数据的话，那么一定要用batch批量操作的api

尽可能减少网络开销次数，可能可以将性能提升数倍，甚至数十倍，非常非常之重要

## 分布式文件系统-上机动手实战演练 bulk 批量增删改

1、bulk语法

bulk 语法后面会有很多样式的 json 串。

POST /\_bulk

{ "delete": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_index": "test\_index", "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

每一个操作要两个json串，语法如下：

{"action": {"metadata"}}

{"data"}

举例，比如你现在要创建一个文档，放bulk里面，看起来会是这样子的：

{"index": {"\_index": "test\_index", "\_type", "test\_type", "\_id": "1"}}

{"test\_field1": "test1", "test\_field2": "test2"}

有哪些类型的操作可以执行呢？

（1）delete：删除一个文档，只要1个json串就可以了

（2）create：PUT /index/type/id/\_create，强制创建

（3）index：普通的put操作，可以是创建文档，也可以是全量替换文档

（4）update：执行的partial update操作

bulk api对json的语法，有严格的要求，每个json串不能换行，只能放一行，同时一个json串和一个json串之间，必须有一个换行

执行出错的结果：

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "json\_e\_o\_f\_exception",  "reason": "Unexpected end-of-input: expected close marker for Object (start marker at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 1])\n at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 3]"  }  ],  "type": "json\_e\_o\_f\_exception",  "reason": "Unexpected end-of-input: expected close marker for Object (start marker at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 1])\n at [Source: org.elasticsearch.transport.netty4.ByteBufStreamInput@5a5932cd; line: 1, column: 3]"  },  "status": 500  } |

执行成功后的结果：

|  |
| --- |
| {  "took": 41,  "errors": true,  "items": [  {  "delete": {  "found": true,  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "10",  "\_version": 3,  "result": "deleted",  "\_shards": {  "total": 2,  "successful": 1,  "failed": 0  },  "status": 200  }  },  {  "create": {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "3",  "\_version": 1,  "result": "created",  "\_shards": {  "total": 2,  "successful": 1,  "failed": 0  },  "created": true,  "status": 201  }  },  {  "create": {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "2",  "status": 409,  "error": {  "type": "version\_conflict\_engine\_exception",  "reason": "[test\_type][2]: version conflict, document already exists (current version [1])",  "index\_uuid": "6m0G7yx7R1KECWWGnfH1sw",  "shard": "2",  "index": "test\_index"  }  }  },  {  "index": {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "4",  "\_version": 1,  "result": "created",  "\_shards": {  "total": 2,  "successful": 1,  "failed": 0  },  "created": true,  "status": 201  }  },  {  "index": {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "2",  "\_version": 2,  "result": "updated",  "\_shards": {  "total": 2,  "successful": 1,  "failed": 0  },  "created": false,  "status": 200  }  },  {  "update": {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "1",  "\_version": 3,  "result": "updated",  "\_shards": {  "total": 2,  "successful": 1,  "failed": 0  },  "status": 200  }  }  ]  } |

**bulk操作中，任意一个操作失败，是不会影响其他的操作的，但是在返回结果里，会告诉你异常日志**

POST /test\_index/\_bulk

{ "delete": { "\_type": "test\_type", "\_id": "3" }}

{ "create": { "\_type": "test\_type", "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { "\_type": "test\_type" }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_type": "test\_type", "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_type": "test\_type", "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

POST /test\_index/test\_type/\_bulk

{ "delete": { "\_id": "3" }}

{ "create": { "\_id": "12" }}

{ "test\_field": "test12" }

{ "index": { }}

{ "test\_field": "auto-generate id test" }

{ "index": { "\_id": "2" }}

{ "test\_field": "replaced test2" }

{ "update": { "\_id": "1", "\_retry\_on\_conflict" : 3} }

{ "doc" : {"test\_field2" : "bulk test1"} }

2、bulk size最佳大小

bulk request会加载到内存里，如果太大的话，性能反而会下降，因此需要反复尝试一个最佳的bulk size。一般从**1000~5000**条数据开始，尝试逐渐增加。另外，如果看大小的话，最好是在**5~15MB**之间。

## 阶段性总结

1~8讲：快速入门了一下，最基本的原理，最基本的操作

9~13讲：在入门之后，对ES的分布式的基本原理，进行了相对深入一些的剖析

14~27讲：围绕着document这个东西，进行操作，进行讲解和分析

## ES的 Document 原理相关

### 分布式文件系统-ES 分布式文件系统概览

1、什么是distributed document store

到目前为止，你觉得你在学什么东西，给大家一个直观的感觉，好像已经知道了es是分布式的，包括一些基本的原理，然后花了不少时间在学习document本身相关的操作，增删改查。一句话点出来，给大家归纳总结一下，其实我们应该思考一下，es的一个最最核心的功能，已经被我们相对完整的讲完了。

Elasticsearch在跑起来以后，其实起到的第一个最核心的功能，就是一个**分布式的文档数据存储系统**。ES是分布式的。文档数据存储系统。文档数据，存储系统。

文档数据：es可以存储和操作json文档类型的数据，而且这也是es的核心数据结构。

存储系统：es可以对json文档类型的数据进行存储，查询，创建，更新，删除，等等操作。其实已经起到了一个什么样的效果呢？其实ES满足了这些功能，就可以说已经是一个NoSQL的存储系统了。

围绕着document在操作，其实就是把es当成了一个NoSQL存储引擎，一个可以存储文档类型数据的存储系统，在操作里面的document。

es可以作为一个分布式的文档存储系统，所以说，我们的应用系统，是不是就可以基于这个概念，去进行相关的应用程序的开发了。

什么类型的应用程序呢？

（1）数据量较大，es的分布式本质，可以帮助你快速进行扩容，承载大量数据

（2）数据结构灵活多变，随时可能会变化，而且数据结构之间的关系，非常复杂，如果我们用传统数据库，那是不是很坑，因为要面临大量的表

（3）对数据的相关操作，较为简单，比如就是一些简单的增删改查，用我们之前讲解的那些document操作就可以搞定

（4）NoSQL数据库，适用的也是类似于上面的这种场景

举个例子，比如说像一些网站系统，或者是普通的电商系统，博客系统，面向对象概念比较复杂，但是作为终端网站来说，没什么太复杂的功能，就是一些简单的CRUD操作，而且数据量可能还比较大。这个时候选用ES这种NoSQL型的数据存储，比传统的复杂的功能务必强大的支持SQL的关系型数据库，更加合适一些。无论是性能，还是吞吐量，可能都会更好。

### 分布式文件系统- document 路由原理

源码：PlainOperationRouting

（1）document路由到shard上是什么意思？

（2）路由算法：shard = hash(routing) % number\_of\_primary\_shards

举个例子，一个index有3个primary shard，P0，P1，P2

每次增删改查一个document的时候，都会带过来一个routing number，默认就是这个document的\_id（可能是手动指定，也可能是自动生成）

routing = \_id，假设\_id=1

会将这个routing值，传入一个hash函数中，产出一个routing值的hash值，hash(routing) = 21

然后将hash函数产出的值对这个index的primary shard的数量求余数，21 % 3 = 0

就决定了，这个document就放在P0上。

决定一个document在哪个shard上，最重要的一个值就是routing值，默认是\_id，也可以手动指定，相同的routing值，每次过来，从hash函数中，产出的hash值一定是相同的

无论hash值是几，无论是什么数字，对number\_of\_primary\_shards求余数，结果一定是在0~number\_of\_primary\_shards-1之间这个范围内的。0,1,2。

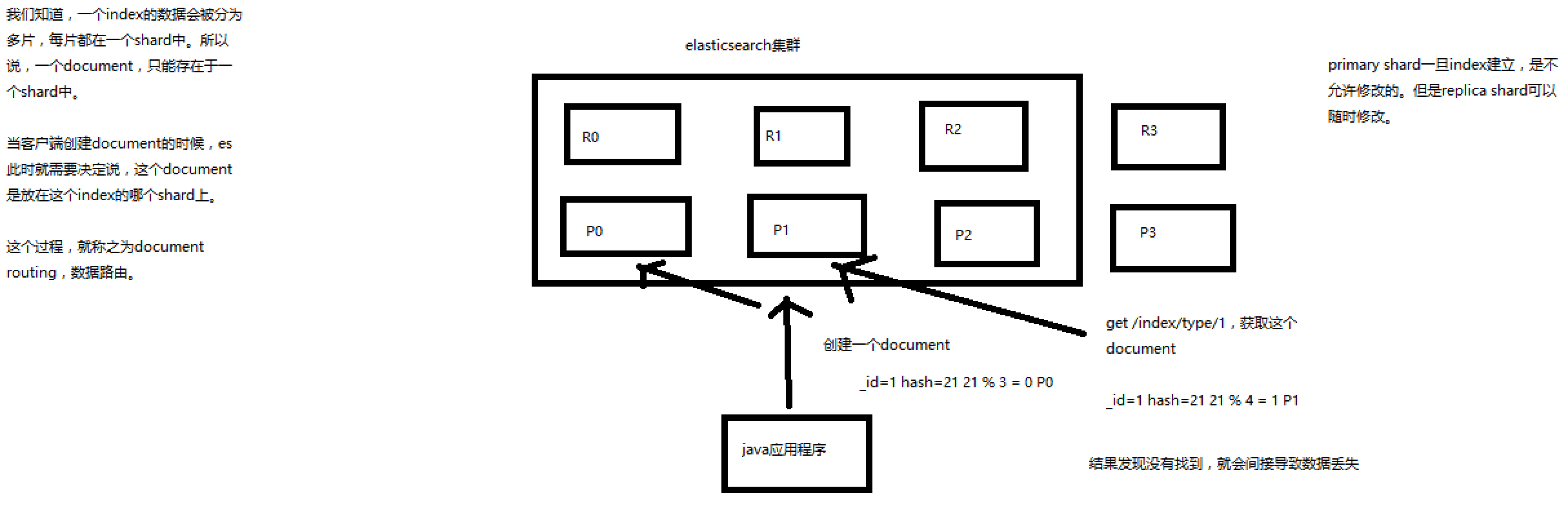
（3）\_id or custom routing value

默认的routing就是\_id

也可以在发送请求的时候，手动指定一个routing value，比如说put /index/type/id?routing=user\_id

手动指定routing value是很有用的，可以保证说，某一类document一定被路由到一个shard上去，那么在后续进行应用级别的负载均衡，以及提升批量读取的性能的时候，是很有帮助的

（4）primary shard数量不可变的谜底

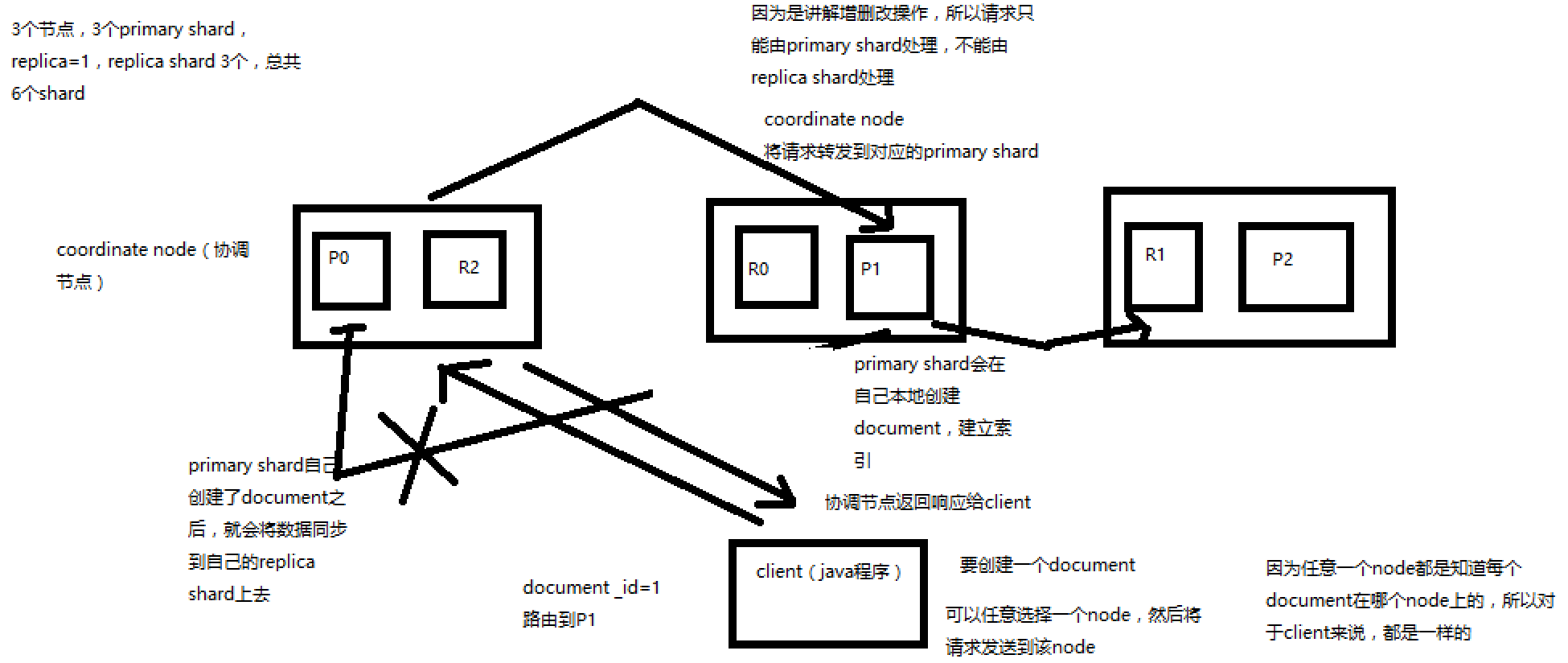


### 分布式文件系统-document 增删改内部原理图揭秘

（1）客户端选择一个node发送请求过去，这个node就是coordinating node（协调节点）

（2）coordinating node，对document进行路由，将请求转发给对应的node（有primary shard）

（3）实际的node上的primary shard处理请求，然后将数据同步到replica node

（4）coordinating node，如果发现primary node和所有replica node都搞定之后，就返回响应结果给客户端

### 分布式文件系统-图解写一致性原理以及 quorum 机制深入剖析

参考资料：

Elasticsearch 写操作剖析：https://www.cnblogs.com/hapjin/p/9821073.html

|  |
| --- |
| **ES官方文档链接以及源码记录**  2.x 版本：  https://www.elastic.co/guide/en/elasticsearch/reference/2.4/docs-index\_.html#index-wait-for-active-shards  源码记录：  1. 在此处定义：ReplicationRequest -> consistencyLevel  2. 如在RestInexAction里的handleRequest获取 consistency 参数。并进行设置参数  3. 在 TransportReplicationAction 里面的checkWriteConsistency方法里面进行判断是否等待需要的情况。并返回相应的值。 |
| 5.x 版本：  https://www.elastic.co/guide/en/elasticsearch/reference/5.6/docs-index\_.html#index-wait-for-active-shards  源码记录：  1. 在此处定义：ReplicationRequest -> waitForActiveShards  2. 如在RestInexAction里的prepareRequest获取 wait\_for\_active\_shards参数。并进行设置参数  3. 在ReplicationOperation 里面的checkActiveShardCount方法里面进行判断是否等待需要的情况。并返回相应的值。 |

笔记记录：

（1）consistency，one（primary shard），all（all shard），quorum（default）

我们在发送任何一个增删改操作的时候，比如说put /index/type/id，都可以带上一个consistency参数，指明我们想要的写一致性是什么？

put /index/type/id?consistency=quorum

**one：**要求我们这个写操作，只要有一个primary shard是active活跃可用的，就可以执行

**all：**要求我们这个写操作，必须所有的primary shard和replica shard都是活跃的，才可以执行这个写操作

**quorum：**默认的值，要求所有的shard中，必须是大部分的shard都是活跃的，可用的，才可以执行这个写操作

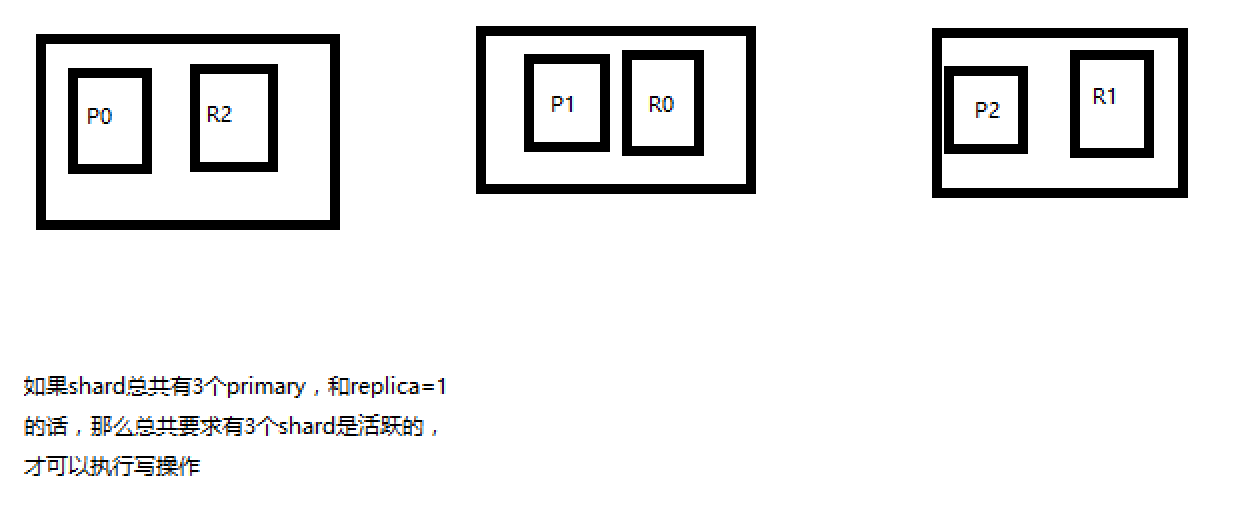
（2）quorum机制，写之前必须确保大多数shard都可用，int( (primary + number\_of\_replicas) / 2 ) + 1，当number\_of\_replicas>1时才生效

quroum = int( (primary + number\_of\_replicas) / 2 ) + 1

举个例子，3个primary shard，number\_of\_replicas=1，总共有3 + 3 \* 1 = 6个shard

quorum = int( (3 + 1) / 2 ) + 1 = 3

所以，要求6个shard中至少有3个shard是active状态的，才可以执行这个写操作



（3）如果节点数少于quorum数量，可能导致quorum不齐全，进而导致无法执行任何写操作

3个primary shard，replica=1，要求至少3个shard是active，3个shard按照之前学习的shard&replica机制，必须在不同的节点上，如果说只有2台机器的话，是不是有可能出现说，3个shard都没法分配齐全，此时就可能会出现写操作无法执行的情况

es提供了一种特殊的处理场景，就是说当number\_of\_replicas>1时才生效，因为假如说，你就一个primary shard，replica=1，此时就2个shard

(1 + 1 / 2) + 1 = 2，要求必须有2个shard是活跃的，但是可能就1个node，此时就1个shard是活跃的，如果你不特殊处理的话，导致我们的单节点集群就无法工作

（4）quorum不齐全时，wait，默认1分钟，timeout，100，30s

等待期间，期望活跃的shard数量可以增加，最后实在不行，就会timeout

我们其实可以在写操作的时候，加一个timeout参数，比如说put /index/type/id?timeout=30，这个就是说自己去设定quorum不齐全的时候，es的timeout时长，可以缩短，也可以增长

### 分布式文件系统-document 查询内部原理图揭秘

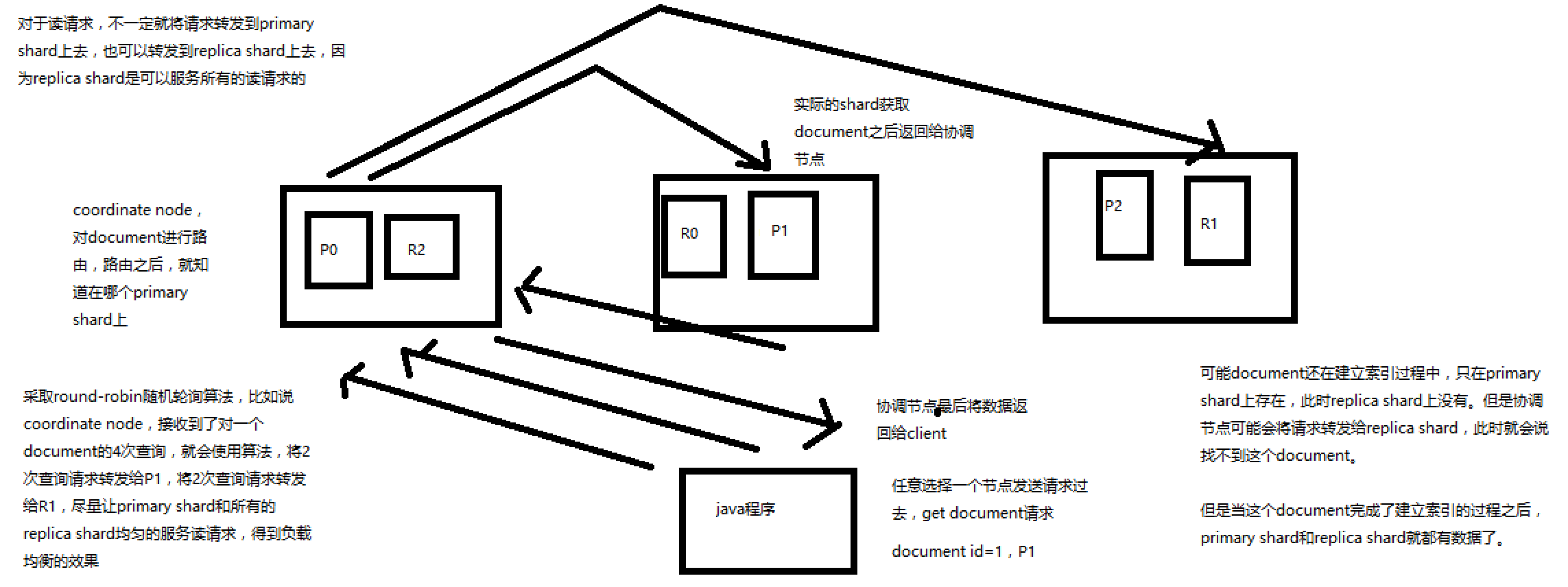
1、客户端发送请求到任意一个node，成为coordinate node

2、coordinate node对document进行路由，将请求转发到对应的node，此时会使用round-robin随机轮询算法，在primary shard以及其所有replica中随机选择一个，让读请求负载均衡

3、接收请求的node返回document给coordinate node

4、coordinate node返回document给客户端

5、特殊情况：document如果还在建立索引过程中，可能只有primary shard有，任何一个replica shard都没有，此时可能会导致无法读取到document，但是document完成索引建立之后，primary shard和replica shard就都有了



## 分布式文件系统-bulk api 的奇特 json 格式与底层性能优化关系大揭秘

bulk api奇特的json格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

[{

"action": {

},

"data": {

}

}]

1、bulk中的每个操作都可能要转发到不同的node的shard去执行

2、如果采用比较良好的json数组格式

允许任意的换行，整个可读性非常棒，读起来很爽，es拿到那种标准格式的json串以后，要按照下述流程去进行处理

（1）将json数组解析为JSONArray对象，这个时候，整个数据，就会在内存中出现一份一模一样的拷贝，一份数据是json文本，一份数据是JSONArray对象

（2）解析json数组里的每个json，对每个请求中的document进行路由

（3）为路由到同一个shard上的多个请求，创建一个请求数组

（4）将这个请求数组序列化

（5）将序列化后的请求数组发送到对应的节点上去

3、耗费更多内存，更多的jvm gc开销

我们之前提到过bulk size最佳大小的那个问题，一般建议说在几千条那样，然后大小在10MB左右，所以说，可怕的事情来了。假设说现在100个bulk请求发送到了一个节点上去，然后每个请求是10MB，100个请求，就是1000MB = 1GB，然后每个请求的json都copy一份为jsonarray对象，此时内存中的占用就会翻倍，就会占用2GB的内存，甚至还不止。因为弄成jsonarray之后，还可能会多搞一些其他的数据结构，2GB+的内存占用。

占用更多的内存可能就会积压其他请求的内存使用量，比如说最重要的搜索请求，分析请求，等等，此时就可能会导致其他请求的性能急速下降

另外的话，占用内存更多，就会导致java虚拟机的垃圾回收次数更多，跟频繁，每次要回收的垃圾对象更多，耗费的时间更多，导致es的java虚拟机停止工作线程的时间更多

4、现在的奇特格式

{"action": {"meta"}}\n

{"data"}\n

{"action": {"meta"}}\n

{"data"}\n

（1）不用将其转换为json对象，不会出现内存中的相同数据的拷贝，直接按照换行符切割json

（2）对每两个一组的json，读取meta，进行document路由

（3）直接将对应的json发送到node上去

5、最大的优势在于，不需要将json数组解析为一个JSONArray对象，形成一份大数据的拷贝，浪费内存空间，尽可能地保证性能

## 初识搜索引擎 - 相关知识点

### search 结果解析(search timeout机制揭秘)

1、我们如果发出一个搜索请求的话，会拿到一堆搜索结果，本节课，我们来讲解一下，这个搜索结果里的各种数据，都代表了什么含义

GET /\_search

命令的到的结果数据会有很多，进行裁剪后的数据案例如下所示：

|  |
| --- |
| {  "took": 6,  "timed\_out": false,  "\_shards": {  "total": 6,  "successful": 6,  "failed": 0  },  "hits": {  "total": 10,  "max\_score": 1,  "hits": [  {  "\_index": ".kibana",  "\_type": "config",  "\_id": "5.2.0",  "\_score": 1,  "\_source": {  "buildNum": 14695  }  }  ]  }  } |

took：整个搜索请求花费了多少毫秒

hits.total：本次搜索，返回了几条结果

hits.max\_score：本次搜索的所有结果中，最大的相关度分数是多少，每一条document对于search的相关度，越相关，\_score分数越大，排位越靠前

hits.hits：默认查询前10条数据，完整数据，\_score降序排序

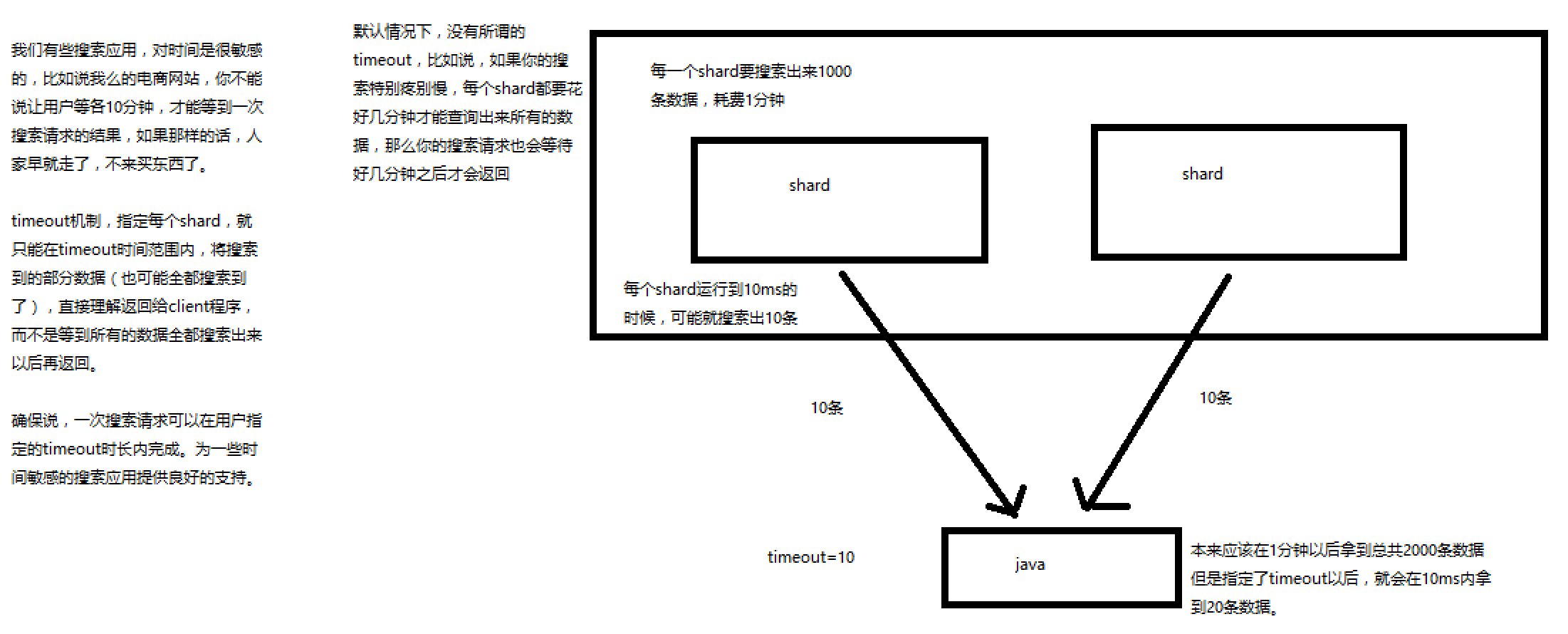
shards：shards fail的条件（primary和replica全部挂掉），不影响其他shard。默认情况下来说，一个搜索请求，会打到一个index的所有primary shard上去，当然了，每个primary shard都可能会有一个或多个replic shard，所以请求也可以到primary shard的其中一个replica shard上去。

timeout：默认无timeout，latency平衡completeness，手动指定timeout，timeout查询执行机制

2、我们来讲解一下，搜索的timeout机制，底层的原理，画图讲解

timeout=10ms，timeout=1s，timeout=1m

GET /\_search?timeout=10m



### multi-index&multi-type 搜索模式解析以及搜索原理初步图解

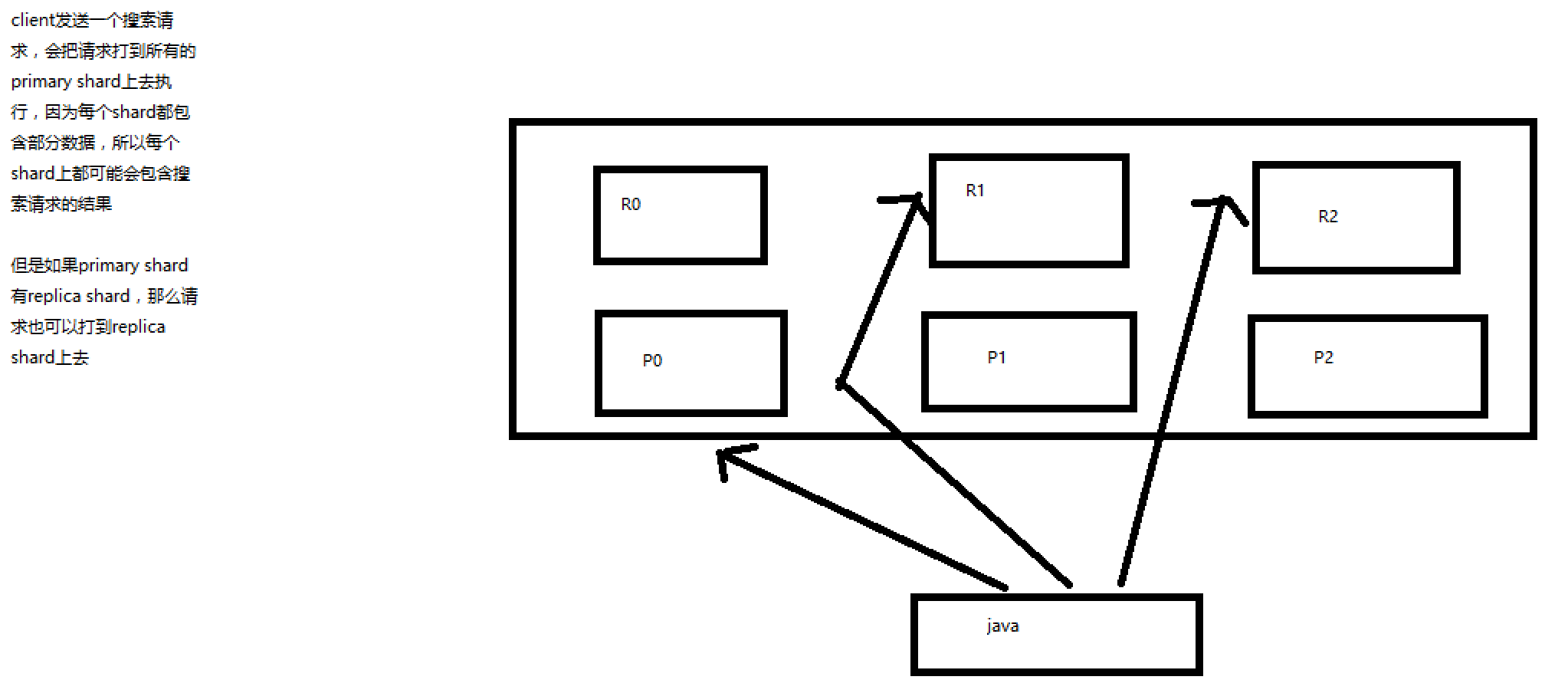
1、multi-index和multi-type搜索模式

告诉你如何一次性搜索多个index和多个type下的数据

|  |  |
| --- | --- |
| **语法格式** | **语法解释** |
| /\_search | 所有索引，所有type下的所有数据都搜索出来 |
| /{index1}/\_search | 指定一个index，搜索其下所有type的数据 |
| /{index1,index2}/\_search | 同时搜索两个index下的数据 |
| /\*1,\*2/\_search | 按照通配符去匹配多个索引 |
| /{index1}/{type1}/\_search | 搜索一个index下指定的type的数据 |
| /{index1/{type1,type2}/\_search | 可以搜索一个index下多个type的数据 |
| /{index1,index2}/{type1,type2}/\_search | 搜索多个index下的多个type的数据 |
| /\_all/{type1,type2}/\_search | \_all，可以代表搜索所有index下的指定type的数据 |

2、初步图解一下简单的搜索原理

client 发送一个搜索请求，会把请求打到所有的 primary shard 上去执行，因为每个 shard 都包含部分数据，所以每个 shard 上都可能会包含搜索请求的结果。但是如果 primary shard 有 replica shard，那么请求也可以到 replica shard 上去。

搜索原理初步图解

### 分页搜索以及 deep paging 性能问题深度图解

1、讲解如何使用es进行分页搜索的语法

size：一页显示几条数据

from：从第几条开始显示

一般是传 pageSize（一页的条数），还有 pageNum（第几页）

size = pageSize

from = (pageNum - 1) x pageSize

命令格式案例：

GET /\_search?size=10

GET /\_search?size=10&from=0

GET /\_search?size=10&from=20

分页的上机实验

GET /test\_index/test\_type/\_search

"hits": {

"total": 9,

"max\_score": 1,

我们假设将这9条数据分成3页，每一页是3条数据，来实验一下这个分页搜索的效果

GET /test\_index/test\_type/\_search?from=0&size=3

结果：

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 9,  "max\_score": 1,  "hits": [  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "8",  "\_score": 1,  "\_source": {  "test\_field": "test client 2"  }  },  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "6",  "\_score": 1,  "\_source": {  "test\_field": "tes test"  }  },  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "4",  "\_score": 1,  "\_source": {  "test\_field": "test4"  }  }  ]  }  } |

第一页：id=8,6,4

GET /test\_index/test\_type/\_search?from=3&size=3

第二页：id=2,自动生成,7

GET /test\_index/test\_type/\_search?from=6&size=3

第三页：id=1,11,3

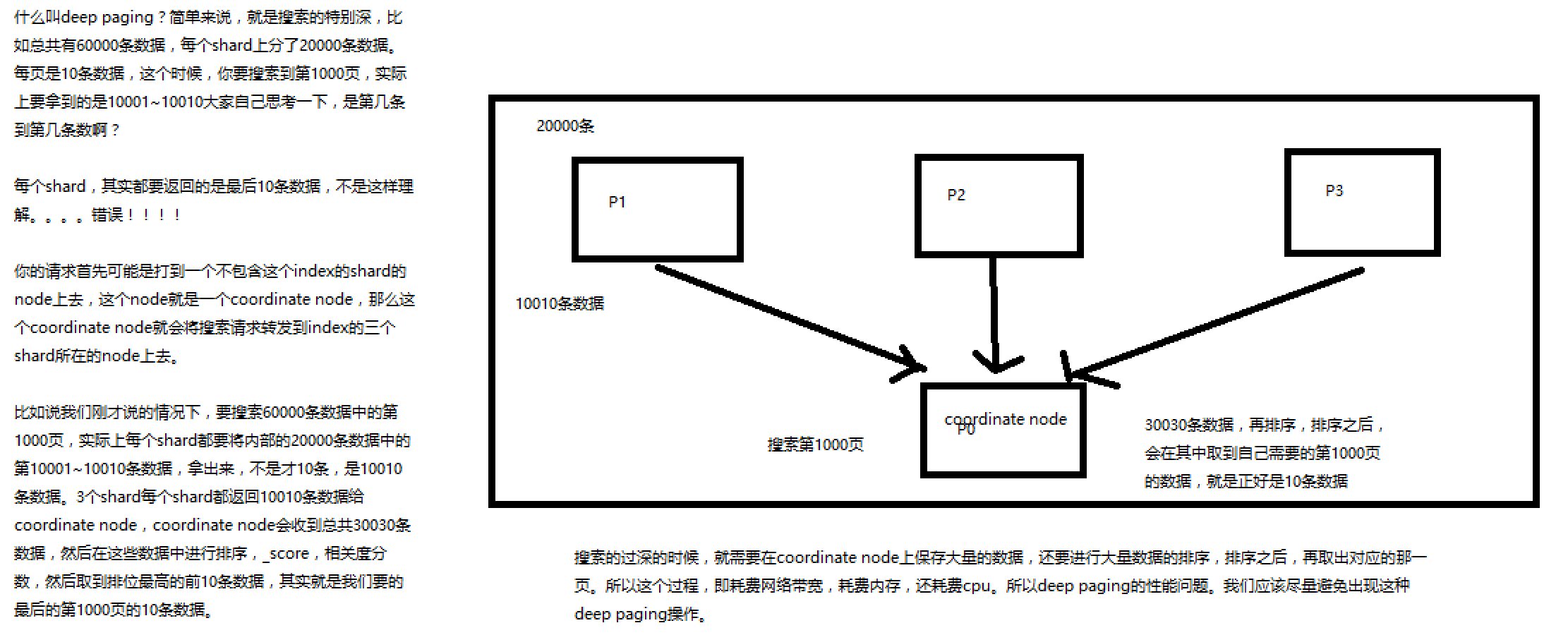
2、什么是deep paging问题？为什么会产生这个问题，它的底层原理是什么？

deep paging性能问题，以及原理深度图解揭秘，很高级的知识点。

6w 条数据，3 个 shard，每个 shard 上分2w 条数据。情景：每页 10 条数据，我去第 1000 页的数据。实际就是滴 10001~10010 的数据。

coordinate node 拿到每个shard 的10010的数据，总共 30030 条数据，然后进行排序，打分。最后返回前 10 条数据。

从上面的过程可以看出来，coordinate node 承载了很多的数据，并且需要大量的排序，并且 shard 数据汇总到 coordinate node，需要耗费大量的网络资源。所以需要避免 deep paging 问题。



### query string search 语法以及\_all metadata原理揭秘

1、query string基础语法

GET /test\_index/test\_type/\_search?q=test\_field:test

GET /test\_index/test\_type/\_search?q=+test\_field:test

GET /test\_index/test\_type/\_search?q=-test\_field:test

一个是掌握q=field:search content的语法，还有一个是掌握+和-的含义

① 不写就是 field 字段包含 search content 内容的全部搜索出来

② +：表示必须包含 search content，跟不写是一样的情况

③ -：表示不包含 search content 内容的搜索出来

2、\_all metadata的原理和作用

GET /test\_index/test\_type/\_search?q=test

直接可以搜索所有的field，不指定 filed 字段，直接搜索q=后面的内容。任意一个field包含指定的关键字就可以搜索出来。

疑问：我们在进行中搜索的时候，难道是对document中的每一个field都进行一次搜索吗？不是的

解答：es中的\_all元数据，在建立索引的时候，我们插入一条document，它里面包含了多个field，此时，es会自动将多个field的值，全部用字符串的方式串联起来，变成一个长的字符串，作为\_all field的值，同时建立索引。

后面如果在搜索的时候，没有对某个field指定搜索，就默认搜索\_all field，其中是包含了所有field的值的

举个例子：

{

"name": "jack",

"age": 26,

"email": "jack@sina.com",

"address": "guamgzhou"

}

"jack 26 jack@sina.com guangzhou"，作为这一条document的\_all field的值，同时进行分词后建立对应的倒排索引

！！！生产环境不使用！！！

### mapping 的引入

插入几条数据，让es自动为我们建立一个索引

PUT /website/article/1

{

"post\_date": "2017-01-01",

"title": "my first article",

"content": "this is my first article in this website",

"author\_id": 11400

}

PUT /website/article/2

{

"post\_date": "2017-01-02",

"title": "my second article",

"content": "this is my second article in this website",

"author\_id": 11400

}

PUT /website/article/3

{

"post\_date": "2017-01-03",

"title": "my third article",

"content": "this is my third article in this website",

"author\_id": 11400

}

尝试各种搜索

GET /website/article/\_search?q=2017 3条结果

GET /website/article/\_search?q=2017-01-01 3条结果，结果很诧异，应该是一条啊

GET /website/article/\_search?q=post\_date:2017-01-01 1条结果

GET /website/article/\_search?q=post\_date:2017 1条结果，结果也很差异，应该是三条

查看es自动建立的mapping，带出什么是mapping的知识点

自动或手动为index中的type建立的一种数据结构和相关配置，简称为mapping，不指定 mapping 直接创建 index 索引数据化，就称为dynamic mapping，自动为我们建立index，创建type，以及type对应的mapping，mapping中包含了每个field对应的数据类型，以及如何分词等设置

当然，后面会讲解，也可以手动在创建数据之前，先创建index和type，以及type对应的mapping

使用如下命令查看 website 下的 article 的 mapping 映射关系

GET /website/\_mapping/article

结果：

|  |
| --- |
| {  "website": {  "mappings": {  "article": {  "properties": {  "author\_id": {  "type": "long"  },  "content": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "post\_date": {  "type": "date"  },  "title": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  }  }  }  }  }  } |

这里简单说一下原因：搜索结果为什么不一致，因为es自动建立mapping的时候，设置了不同的field不同的data type。不同的data type的分词、搜索等行为是不一样的。所以出现了\_all field和post\_date field的搜索表现完全不一样。

### 精确匹配与全文检索的对比分析

1、exact value

2017-01-01，exact value，搜索的时候，必须输入2017-01-01，才能搜索出来

如果你输入一个01，是搜索不出来的

2、full text

（1）缩写 vs. 全程：cn vs. china

（2）格式转化：like liked likes

（3）大小写：Tom vs tom

（4）同义词：like vs love

2017-01-01，2017 01 01，搜索2017，或者01，都可以搜索出来

china，搜索cn，也可以将china搜索出来

likes，搜索like，也可以将likes搜索出来

Tom，搜索tom，也可以将Tom搜索出来

like，搜索love，同义词，也可以将like搜索出来

就不是说单纯的只是匹配完整的一个值，而是可以对值进行拆分词语后（分词）进行匹配，也可以通过缩写、时态、大小写、同义词等进行匹配

### 倒排索引核心原理快速揭秘

下面有两句话，doc1 和 doc2，我们拿这两句话举例子来说明倒排索引的原理

doc1：I really liked my small dogs, and I think my mom also liked them.

doc2：He never liked any dogs, so I hope that my mom will not expect me to liked him.

STEP1：分词，初步的倒排索引的建立。将单词一空格分开，如果有的就标记为 \* 符号。

|  |  |  |
| --- | --- | --- |
| word | doc1 | doc2 |
| I | \* | \* |
| really | \* |  |
| liked | \* | \* |
| my | \* | \* |
| small | \* |  |
| dogs | \* |  |
| and | \* |  |
| think | \* |  |
| mom | \* | \* |
| also | \* |  |
| them | \* |  |
| He |  | \* |
| never |  | \* |
| any |  | \* |
| so |  | \* |
| hope |  | \* |
| that |  | \* |
| will |  | \* |
| not |  | \* |
| expect |  | \* |
| me |  | \* |
| to |  | \* |
| him |  | \* |

上面演示了一下倒排索引最简单的建立的一个过程

这里我们进行搜索mother like little dog，这个搜索的也会进行拆分为四个单词，然后去上面的表格里面去进行匹配，但是一个都搜不到，所以不可能有任何结果。

这个是不是我们想要的搜索结果？？？绝对不是，因为在我们看来，mother和mom有区别吗？同义词，都是妈妈的意思。like和liked有区别吗？没有，都是喜欢的意思，只不过一个是现在时，一个是过去时。little和small有区别吗？同义词，都是小小的。dog和dogs有区别吗？狗，只不过一个是单数，一个是复数。

normalization，建立倒排索引的时候，会执行一个操作，也就是说对拆分出的各个单词进行相应的处理，以提升后面搜索的时候能够搜索到相关联的文档的概率

时态的转换，单复数的转换，同义词的转换，大小写的转换

mom ―> mother

liked ―> like

small ―> little

dogs ―> dog

下面进行重新建立倒排索引，加入normalization，再次用mother liked little dog搜索，就可以搜索到了

|  |  |  |  |
| --- | --- | --- | --- |
| word | doc1 | doc2 |  |
| I | \* | \* |  |
| really | \* |  |  |
| liked | \* | \* | liked --> like |
| my | \* | \* |  |
| small | \* |  | small --> little |
| dogs | \* |  | dogs --> dog |
| and | \* |  |  |
| think | \* |  |  |
| mom | \* | \* |  |
| also | \* |  |  |
| them | \* |  |  |
| He |  | \* |  |
| never |  | \* |  |
| any |  | \* |  |
| so |  | \* |  |
| hope |  | \* |  |
| that |  | \* |  |
| will |  | \* |  |
| not |  | \* |  |
| expect |  | \* |  |
| me |  | \* |  |
| to |  | \* |  |
| him |  | \* |  |

这里在进行搜索mother like little dog，他会进行分词、normalization的处理。这样就可以搜到了结果

mother --> mom

like --> like

little --> little

dog --> dog

结果就是：doc1和doc2都会搜索出来

doc1：I really liked my small dogs, and I think my mom also liked them.

doc2：He never liked any dogs, so I hope that my mom will not expect me to liked him.

### 分词器的内部组成到底是什么，以及内置分词器的介绍

1、什么是分词器

切分词语，normalization（提升recall召回率）

给你一段句子，然后将这段句子拆分成一个一个的单个的单词，同时对每个单词进行normalization（时态转换，单复数转换），分词器

recall，召回率：搜索的时候，增加能够搜索到的结果的数量

下面是分词的一个清洗过程：

character filter：在一段文本进行分词之前，先进行预处理，比如说最常见的就是，过滤html标签（<span>hello<span> --> hello），& --> and（I&you --> I and you）

tokenizer：分词，hello you and me --> hello, you, and, me

token filter：lowercase，stop word，synonymom，dogs --> dog，liked --> like，Tom --> tom，a/the/an --> 干掉，mother --> mom，small --> little

一个分词器，很重要，将一段文本进行各种处理，最后处理好的结果才会拿去建立倒排索引

2、内置分词器的介绍

Set the shape to semi-transparent by calling set\_trans(5)

**standard analyzer**：set, the, shape, to, semi, transparent, by, calling, set\_trans, 5（默认的是standard）

**simple analyzer**：set, the, shape, to, semi, transparent, by, calling, set, trans

**whitespace analyzer**：Set, the, shape, to, semi-transparent, by, calling, set\_trans(5)

**language analyzer**（特定的语言的分词器，比如说，english，英语分词器）：set, shape, semi, transpar, call, set\_tran, 5

### query string 的分词以及 mapping 引入案例遗留问题的大揭秘

1、query string分词

query string必须以和index建立时相同的analyzer进行分词

query string对exact value和full text的区别对待

date：exact value

\_all：full text

比如我们有一个document，其中有一个field，包含的value是：hello you and me，建立倒排索引

我们要搜索这个document对应的index，搜索文本是hell me，这个搜索文本就是query string

query string，默认情况下，es会使用它对应的field建立倒排索引时相同的分词器去进行分词，分词和normalization，只有这样，才能实现正确的搜索

我们建立倒排索引的时候，将dogs --> dog，结果你搜索的时候，还是一个dogs，那不就搜索不到了吗？所以搜索的时候，那个dogs也必须变成dog才行。才能搜索到。

知识点：不同类型的field，可能有的就是full text，有的就是exact value

post\_date，date：exact value

\_all：full text，分词，normalization

2、mapping引入案例遗留问题大揭秘

GET /\_search?q=2017

上面的命令搜索的是\_all field，document所有的field都会拼接成一个大串，进行分词，如下面所示

2017-01-02 my second article this is my second article in this website 11400

对 2017-01-02 分词后的情况如下表所示。

doc1 doc2 doc3

2017 \* \* \*

01 \*

02 \*

03 \*

**情况一：**对于\_all，2017，自然会搜索到3个docuemnt

对于具体的搜索，如下所示：

GET /\_search?q=2017-01-01

**情况二：**\_all，2017-01-01，query string会用跟建立倒排索引一样的分词器去进行分词

2017

01

01

**情况三：**GET /\_search?q=post\_date:2017-01-01

date，会作为exact value去建立索引，所以结果是 1 条。如下所示：

doc1 doc2 doc3

2017-01-01 \*

2017-01-02 \*

2017-01-03 \*

**情况四：**GET /\_search?q=post\_date:2017，这个在这里不讲解，因为是es 5.2以后做的一个优化

3、测试分词器

GET /\_analyze

{

"analyzer": "standard",

"text": "Text to analyze"

}

### mapping 再次回炉透彻理解

（1）往es里面直接插入数据，es会自动建立索引，同时建立type以及对应的mapping

（2）mapping中就自动定义了每个field的数据类型

（3）不同的数据类型（比如说text和date），可能有的是exact value，有的是full text

（4）exact value，在建立倒排索引的时候，分词的时候，是将整个值一起作为一个关键词建立到倒排索引中的；full text，会经历各种各样的处理，分词，normaliztion（时态转换，同义词转换，大小写转换），才会建立到倒排索引中

（5）同时呢，exact value和full text类型的field就决定了，在一个搜索过来的时候，对exact value field或者是full text field进行搜索的行为也是不一样的，会跟建立倒排索引的行为保持一致；比如说exact value搜索的时候，就是直接按照整个值进行匹配，full text query string，也会进行分词和normalization再去倒排索引中去搜索

（6）可以用es的dynamic mapping，让其自动建立mapping，包括自动设置数据类型；也可以提前手动创建index和type的mapping，自己对各个field进行设置，包括数据类型，包括索引行为，包括分词器，等等

mapping，就是index的type的元数据，每个type都有一个自己的mapping，决定了数据类型，建立倒排索引的行为，还有进行搜索的行为

### mapping 数据类型以及 dynamic mapping

1、核心的数据类型

string

byte，short，integer，long

float，double

boolean

date

2、动态推测数据类型，如下面的情况的一些例子

true or false --> boolean

123 --> long

123.45 --> double

2017-01-01 --> date

"hello world" --> string/text

3、查看mapping

GET /{index}/\_mapping/{type}

### mapping之手动建立和修改以及定制 string 类型数据是否分词

1、如何建立索引，有如下三个值可以选择

analyzed：分词

not\_analyzed：分词，直接值匹配

no：不分词，也不搜索

2、修改mapping

只能创建index时手动建立mapping，或者新增field mapping，但是不能update field mapping

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "long"

},

"title": {

"type": "text",

"analyzer": "english"

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"publisher\_id": {

"type": "text",

"index": "not\_analyzed"

}

}

}

}

}

PUT /website

{

"mappings": {

"article": {

"properties": {

"author\_id": {

"type": "text"

}

}

}

}

}

上面视图修改 author\_id的 mapping，可以看见出现错误信息。结果如下所示：

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "index\_already\_exists\_exception",  "reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",  "index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",  "index": "website"  }  ],  "type": "index\_already\_exists\_exception",  "reason": "index [website/co1dgJ-uTYGBEEOOL8GsQQ] already exists",  "index\_uuid": "co1dgJ-uTYGBEEOOL8GsQQ",  "index": "website"  },  "status": 400  } |

这新增一个 mapping field，可以看见是可以的。

PUT /website/\_mapping/article

{

"properties" : {

"new\_field" : {

"type" : "keyword",

"index": "not\_analyzed"

}

}

}

3、测试mapping

content 字段分词测试

GET /website/\_analyze

{

"field": "content",

"text": "my-dogs"

}

new\_field 字段分词测试。会报错

GET website/\_analyze

{

"field": "new\_field",

"text": "my dogs"

}

上面的创建的 new\_field是不分词的，现在进行对他分词测试。可看见出现不支持分词的错误信息。结果如下所示：

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "remote\_transport\_exception",  "reason": "[4onsTYV][127.0.0.1:9300][indices:admin/analyze[s]]"  }  ],  "type": "illegal\_argument\_exception",  "reason": "Can't process field [new\_field], Analysis requests are only supported on tokenized fields"  },  "status": 400  } |

### mapping 复杂数据类型以及 object 类型数据底层结构大揭秘

复杂数据类型如下所示：

1、multivalue field

{ "tags": [ "tag1", "tag2" ]}

建立索引时与string是一样的，数据类型不能混

2、empty field

null，[]，[null]

3、object field

示范 objec 条数据类型。添加一条数据如下：

PUT /company/employee/1

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

使用 GET /company/\_mapping/employee命令查看 mapping 数据。可以看见，address 就是object数据类型。

结果：

|  |
| --- |
| {  "company": {  "mappings": {  "employee": {  "properties": {  "address": {  "properties": {  "city": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "country": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "province": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  }  }  },  "age": {  "type": "long"  },  "join\_date": {  "type": "date"  },  "name": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  }  }  }  }  }  } |

{

"address": {

"country": "china",

"province": "guangdong",

"city": "guangzhou"

},

"name": "jack",

"age": 27,

"join\_date": "2017-01-01"

}

这里 adrredss 就是 object 数据类型

object 底层会变成什么样了？大概就是如下格式的样子：

{

"name": [jack],

"age": [27],

"join\_date": [2017-01-01],

"address.country": [china],

"address.province": [guangdong],

"address.city": [guangzhou]

}

如果是复杂些的数据了，如下 json 串是一个数组，然后数组里面的元素又是多个值的。如下：

{

"authors": [

{ "age": 26, "name": "Jack White"},

{ "age": 55, "name": "Tom Jones"},

{ "age": 39, "name": "Kitty Smith"}

]

}

底层数据会存成如下样式的：

{

"authors.age": [26, 55, 39],

"authors.name": [jack, white, tom, jones, kitty, smith]

}

### search api 的基础语法介绍

1、search api的基本语法

GET /search

{}

GET /index1,index2/type1,type2/search

{}

GET /\_search

{

"from": 0,

"size": 10

}

2、http协议中get是否可以带上request body

HTTP协议，一般不允许get请求带上request body，但是因为get更加适合描述查询数据的操作，因此还是这么用了

GET /\_search?from=0&size=10

POST /\_search

{

"from":0,

"size":10

}

碰巧，很多浏览器，或者是服务器，也都支持GET+request body模式

如果遇到不支持的场景，也可以用POST /\_search

### Query DSL快速上手动手实战搜索语法

1、一个例子让你明白什么是Query DSL

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、Query DSL的基本语法

{

QUERY\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

{

QUERY\_NAME: {

FIELD\_NAME: {

ARGUMENT: VALUE,

ARGUMENT: VALUE,...

}

}

}

示例：

GET /test\_index/test\_type/\_search

{

"query": {

"match": {

"test\_field": "test"

}

}

}

3、如何组合多个搜索条件

语法格式：

GET /test\_index/\_search

{

"query": {

"bool": {

"must": { "match": { "name": "tom" }},

"should": [

{ "match": { "hired": true }},

{ "bool": {

"must": { "match": { "personality": "good" }},

"must\_not": { "match": { "rude": true }}

}}

],

"minimum\_should\_match": 1

}

}

}

must：必须满足条件

should：就是满足不满足也可以

must not：必须不满足条件

搜索需求：

\* title必须包含elasticsearch，

\* content可以包含elasticsearch也可以不包含，

\* author\_id必须不为111

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

结果：

|  |
| --- |
| {  "took": 1,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1,  "hits": [  {  "\_index": "website",  "\_type": "article",  "\_id": "1",  "\_score": 1,  "\_source": {  "title": "my elasticsearch article",  "content": "es is very bad",  "author\_id": 110  }  }  ]  }  } |

### filter 与 query 深入对比解密：相关度，性能

1、filter与query示例

先插入几条数据

PUT /company/employee/2

{

"address": {

"country": "china",

"province": "jiangsu",

"city": "nanjing"

},

"name": "tom",

"age": 30,

"join\_date": "2016-01-01"

}

PUT /company/employee/3

{

"address": {

"country": "china",

"province": "shanxi",

"city": "xian"

},

"name": "marry",

"age": 35,

"join\_date": "2015-01-01"

}

搜索请求：年龄必须大于等于30，同时join\_date必须是2016-01-01

GET /company/employee/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"join\_date": "2016-01-01"

}

}

],

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

2、filter与query对比大解密

filter，仅仅只是按照搜索条件过滤出需要的数据而已，不计算任何相关度分数，对相关度没有任何影响

query，会去计算每个document相对于搜索条件的相关度，并按照相关度进行排序

一般来说，如果你是在进行搜索，需要将最匹配搜索条件的数据先返回，那么用query；如果你只是要根据一些条件筛选出一部分数据，不关注其排序，那么用filter

除非是你的这些搜索条件，你希望越符合这些搜索条件的document越排在前面返回，那么这些搜索条件要放在query中；如果你不希望一些搜索条件来影响你的document排序，那么就放在filter中即可

3、filter与query性能

filter，不需要计算相关度分数，不需要按照相关度分数进行排序，同时还有内置的自动cache最常使用filter的数据

query，相反，要计算相关度分数，按照分数进行排序，而且无法cache结果

### query 搜索各种语法-上机动手实战

1、match all

GET /\_search

{

"query": {

"match\_all": {}

}

}

2、match

匹配到的，这里会进行分词。不想 term 是直接匹配整个的。

GET /\_search

{

"query": {

"match": {

"title": "my elasticsearch article"

}

}

}

3、multi match

query 的值使用到多个字段中。

GET /test\_index/test\_type/\_search

{

"query": {

"multi\_match": {

"query": "test",

"fields": ["test\_field", "test\_field1"]

}

}

}

4、range query

range 可以放在 query 里面，也可以放在 filter 里面。

GET /company/employee/\_search

{

"query": {

"range": {

"age": {

"gte": 30

}

}

}

}

5、term query

当成 exact value，就是要相等的才匹配出来。不会进行分词

GET /test\_index/test\_type/\_search

{

"query": {

"term": {

"test\_field": "test hello"

}

}

}

6、terms query

指定多个词，都匹配到

GET /\_search

{

"query": { "terms": { "tag": [ "search", "full\_text", "nosql" ] }}

}

7、exist query（2.x中的查询，现在已经不提供了）

### 多搜索条件组合查询-上机动手实战

案例：

GET /website/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "elasticsearch"

}

}

],

"should": [

{

"match": {

"content": "elasticsearch"

}

}

],

"must\_not": [

{

"match": {

"author\_id": 111

}

}

]

}

}

}

查询语法模板：

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"range": { "date": { "gte": "2014-01-01" }}

}

}

}

可以放什么东西

多组合的时候必须用到：bool，然后后面可以包含下面的四个关键词：

must，must\_not，should，filter

每个子查询都会计算一个document针对它的相关度分数，然后bool综合所有分数，合并为一个分数，当然filter是不会计算分数的

如下就是 must 分数、must\_not分数和 should 分数。然后综合分数得出一个最后的分数。

{

"bool": {

"must": { "match": { "title": "how to make millions" }},

"must\_not": { "match": { "tag": "spam" }},

"should": [

{ "match": { "tag": "starred" }}

],

"filter": {

"bool": {

"must": [

{ "range": { "date": { "gte": "2014-01-01" }}},

{ "range": { "price": { "lte": 29.99 }}}

],

"must\_not": [

{ "term": { "category": "ebooks" }}

]

}

}

}

}

下面直接使用 filter 的方法，加上constant\_score 关键字。

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

}

}

### 定位不合法的搜索以及原因-上机动手实战

使用 explain 来查询搜索语法错误的问题。

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"math": {

"test\_field": "test"

}

}

}

|  |
| --- |
| {  "valid": false,  "error": "org.elasticsearch.common.ParsingException: no [query] registered for [math]"  } |

GET /test\_index/test\_type/\_validate/query?explain

{

"query": {

"match": {

"test\_field": "test"

}

}

}

|  |
| --- |
| {  "valid": true,  "\_shards": {  "total": 1,  "successful": 1,  "failed": 0  },  "explanations": [  {  "index": "test\_index",  "valid": true,  "explanation": "+test\_field:test #(#\_type:test\_type)"  }  ]  } |

一般用在那种特别复杂庞大的搜索下，比如你一下子写了上百行的搜索，这个时候可以先用validate api去验证一下，搜索是否合法

### 定制搜索结果的排序规则-上机动手实战

1、默认排序规则

默认情况下，是按照\_score降序排序的

然而，某些情况下，可能没有有用的\_score，比如说filter

GET /\_search

{

"query" : {

"bool" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

当然，也可以是constant\_score

GET /\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"author\_id" : 1

}

}

}

}

}

2、定制排序规则

使用 sort 定义排序规则。如下是对 join\_date 进行升序排序。

GET /company/employee/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"age": {

"gte": 30

}

}

}

}

},

"sort": [

{

"join\_date": {

"order": "asc"

}

}

]

}

### string field排序问题解密

如果对一个string field进行排序，结果往往不准确，因为分词后是多个单词，再排序就不是我们想要的结果了

通常解决方案是，将一个string field建立两次索引，一个分词，用来进行搜索；一个不分词，用来进行排序

例如：重新建立 website 的索引，首先建立 mapping 关系。如下所示，article 包含了 title、content、post\_date以及 author\_id字段。

对 title 字段进行 fields 索引第二次。这个索引就不分词。完全存储。

PS：fielddata为正排索引。后面会讲到

PUT /website

{

"mappings": {

"article": {

"properties": {

"title": {

"type": "text",

"fields": {

"raw": {

"type": "string",

"index": "not\_analyzed"

}

},

"fielddata": true

},

"content": {

"type": "text"

},

"post\_date": {

"type": "date"

},

"author\_id": {

"type": "long"

}

}

}

}

}

存入数据：

PUT /website/article/1

{

"title": "first article",

"content": "this is my second article",

"post\_date": "2017-01-01",

"author\_id": 110

}

执行查询

GET /website/article/\_search

{

"query": {

"match\_all": {}

},

"sort": [

{

"title.raw": {

"order": "desc"

}

}

]

}

查询出来的结果是：

{

"took": 2,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 3,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "article",

"\_id": "2",

"\_score": 1,

"\_source": {

"title": "first article",

"content": "this is my first article",

"post\_date": "2017-02-01",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "second article",

"content": "this is my second article",

"post\_date": "2017-01-01",

"author\_id": 110

}

},

{

"\_index": "website",

"\_type": "article",

"\_id": "3",

"\_score": 1,

"\_source": {

"title": "third article",

"content": "this is my third article",

"post\_date": "2017-03-01",

"author\_id": 110

}

}

]

}

}

### 相关度评分&TF-IDF算法独家解密

1、算法介绍

relevance score算法，简单来说，就是计算出，一个索引中的文本，与搜索文本，他们之间的关联匹配程度

Elasticsearch使用的是 term frequency/inverse document frequency算法，简称为TF/IDF算法

①：Term frequency：搜索文本中的各个词条在field文本中出现了多少次，出现次数越多，就越相关

搜索请求：hello world

doc1：hello you, and world is very good

doc2：hello, how are you

②：Inverse document frequency：搜索文本中的各个词条在整个索引的所有文档中出现了多少次，出现的次数越多，就越不相关

搜索请求：hello world

doc1：hello, today is very good

doc2：hi world, how are you

比如说，在index中有1万条document，hello这个单词在所有的document中，一共出现了1000次；world这个单词在所有的document中，一共出现了100次

结果：doc2更相关

③：Field-length norm：field长度，field越长，相关度越弱

搜索请求：hello world

doc1：{ "title": "hello article", "content": "babaaba 1万个单词" }

doc2：{ "title": "my article", "content": "blablabala 1万个单词，hi world" }

hello world在整个index中出现的次数是一样多的

结果：doc1更相关，title field更短

2、\_score是如何被计算出来的

使用 explain 关键词查询评分的详细情况

GET /test\_index/test\_type/\_search?explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

结果：

|  |
| --- |
| {  "took": 6,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 4,  "max\_score": 1.595089,  "hits": [  {  "\_shard": "[test\_index][2]",  "\_node": "4onsTYVZTjGvIj9\_spWz2w",  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "20",  "\_score": 1.595089,  "\_source": {  "test\_field": "test hello"  },  "\_explanation": {  "value": 1.595089,  "description": "sum of:",  "details": [  {  "value": 1.595089,  "description": "sum of:",  "details": [  {  "value": 0.58279467,  "description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",  "details": [  {  "value": 0.58279467,  "description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",  "details": [  {  "value": 0.6931472,  "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",  "details": [  {  "value": 2,  "description": "docFreq",  "details": []  },  {  "value": 4,  "description": "docCount",  "details": []  }  ]  },  {  "value": 0.840795,  "description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",  "details": [  {  "value": 1,  "description": "termFreq=1.0",  "details": []  },  {  "value": 1.2,  "description": "parameter k1",  "details": []  },  {  "value": 0.75,  "description": "parameter b",  "details": []  },  {  "value": 1.75,  "description": "avgFieldLength",  "details": []  },  {  "value": 2.56,  "description": "fieldLength",  "details": []  }  ]  }  ]  }  ]  },  {  "value": 1.0122943,  "description": "weight(test\_field:hello in 0) [PerFieldSimilarity], result of:",  "details": [  {  "value": 1.0122943,  "description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",  "details": [  {  "value": 1.2039728,  "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",  "details": [  {  "value": 1,  "description": "docFreq",  "details": []  },  {  "value": 4,  "description": "docCount",  "details": []  }  ]  },  {  "value": 0.840795,  "description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",  "details": [  {  "value": 1,  "description": "termFreq=1.0",  "details": []  },  {  "value": 1.2,  "description": "parameter k1",  "details": []  },  {  "value": 0.75,  "description": "parameter b",  "details": []  },  {  "value": 1.75,  "description": "avgFieldLength",  "details": []  },  {  "value": 2.56,  "description": "fieldLength",  "details": []  }  ]  }  ]  }  ]  }  ]  },  {  "value": 0,  "description": "match on required clause, product of:",  "details": [  {  "value": 0,  "description": "# clause",  "details": []  },  {  "value": 1,  "description": "\*:\*, product of:",  "details": [  {  "value": 1,  "description": "boost",  "details": []  },  {  "value": 1,  "description": "queryNorm",  "details": []  }  ]  }  ]  }  ]  }  },  {  "\_shard": "[test\_index][2]",  "\_node": "4onsTYVZTjGvIj9\_spWz2w",  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "6",  "\_score": 0.58279467,  "\_source": {  "test\_field": "tes test"  },  "\_explanation": {  "value": 0.58279467,  "description": "sum of:",  "details": [  {  "value": 0.58279467,  "description": "sum of:",  "details": [  {  "value": 0.58279467,  "description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",  "details": [  {  "value": 0.58279467,  "description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",  "details": [  {  "value": 0.6931472,  "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",  "details": [  {  "value": 2,  "description": "docFreq",  "details": []  },  {  "value": 4,  "description": "docCount",  "details": []  }  ]  },  {  "value": 0.840795,  "description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",  "details": [  {  "value": 1,  "description": "termFreq=1.0",  "details": []  },  {  "value": 1.2,  "description": "parameter k1",  "details": []  },  {  "value": 0.75,  "description": "parameter b",  "details": []  },  {  "value": 1.75,  "description": "avgFieldLength",  "details": []  },  {  "value": 2.56,  "description": "fieldLength",  "details": []  }  ]  }  ]  }  ]  }  ]  },  {  "value": 0,  "description": "match on required clause, product of:",  "details": [  {  "value": 0,  "description": "# clause",  "details": []  },  {  "value": 1,  "description": "\*:\*, product of:",  "details": [  {  "value": 1,  "description": "boost",  "details": []  },  {  "value": 1,  "description": "queryNorm",  "details": []  }  ]  }  ]  }  ]  }  },  {  "\_shard": "[test\_index][3]",  "\_node": "4onsTYVZTjGvIj9\_spWz2w",  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "7",  "\_score": 0.5565415,  "\_source": {  "test\_field": "test client 2"  },  "\_explanation": {  "value": 0.5565415,  "description": "sum of:",  "details": [  {  "value": 0.5565415,  "description": "sum of:",  "details": [  {  "value": 0.5565415,  "description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",  "details": [  {  "value": 0.5565415,  "description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",  "details": [  {  "value": 0.6931472,  "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",  "details": [  {  "value": 1,  "description": "docFreq",  "details": []  },  {  "value": 2,  "description": "docCount",  "details": []  }  ]  },  {  "value": 0.8029196,  "description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",  "details": [  {  "value": 1,  "description": "termFreq=1.0",  "details": []  },  {  "value": 1.2,  "description": "parameter k1",  "details": []  },  {  "value": 0.75,  "description": "parameter b",  "details": []  },  {  "value": 2.5,  "description": "avgFieldLength",  "details": []  },  {  "value": 4,  "description": "fieldLength",  "details": []  }  ]  }  ]  }  ]  }  ]  },  {  "value": 0,  "description": "match on required clause, product of:",  "details": [  {  "value": 0,  "description": "# clause",  "details": []  },  {  "value": 1,  "description": "\_type:test\_type, product of:",  "details": [  {  "value": 1,  "description": "boost",  "details": []  },  {  "value": 1,  "description": "queryNorm",  "details": []  }  ]  }  ]  }  ]  }  },  {  "\_shard": "[test\_index][1]",  "\_node": "4onsTYVZTjGvIj9\_spWz2w",  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "8",  "\_score": 0.25316024,  "\_source": {  "test\_field": "test client 2"  },  "\_explanation": {  "value": 0.25316024,  "description": "sum of:",  "details": [  {  "value": 0.25316024,  "description": "sum of:",  "details": [  {  "value": 0.25316024,  "description": "weight(test\_field:test in 0) [PerFieldSimilarity], result of:",  "details": [  {  "value": 0.25316024,  "description": "score(doc=0,freq=1.0 = termFreq=1.0\n), product of:",  "details": [  {  "value": 0.2876821,  "description": "idf, computed as log(1 + (docCount - docFreq + 0.5) / (docFreq + 0.5)) from:",  "details": [  {  "value": 1,  "description": "docFreq",  "details": []  },  {  "value": 1,  "description": "docCount",  "details": []  }  ]  },  {  "value": 0.88,  "description": "tfNorm, computed as (freq \* (k1 + 1)) / (freq + k1 \* (1 - b + b \* fieldLength / avgFieldLength)) from:",  "details": [  {  "value": 1,  "description": "termFreq=1.0",  "details": []  },  {  "value": 1.2,  "description": "parameter k1",  "details": []  },  {  "value": 0.75,  "description": "parameter b",  "details": []  },  {  "value": 3,  "description": "avgFieldLength",  "details": []  },  {  "value": 4,  "description": "fieldLength",  "details": []  }  ]  }  ]  }  ]  }  ]  },  {  "value": 0,  "description": "match on required clause, product of:",  "details": [  {  "value": 0,  "description": "# clause",  "details": []  },  {  "value": 1,  "description": "\*:\*, product of:",  "details": [  {  "value": 1,  "description": "boost",  "details": []  },  {  "value": 1,  "description": "queryNorm",  "details": []  }  ]  }  ]  }  ]  }  }  ]  }  } |

3、分析一个document是如何被匹配上的

GET /test\_index/test\_type/6/\_explain

{

"query": {

"match": {

"test\_field": "test hello"

}

}

}

### 内核级知识doc values 初步探秘

搜索的时候，要依靠倒排索引；排序的时候，需要依靠正排索引，看到每个document的每个field，然后进行排序，所谓的正排索引，其实就是doc values

在建立索引的时候，一方面会建立倒排索引，以供搜索用；一方面会建立正排索引，也就是doc values，以供排序，聚合，过滤等操作使用

doc values是被保存在磁盘上的，此时如果内存足够，os会自动将其缓存在内存中，性能还是会很高；如果内存不足够，os会将其写入磁盘上。使用的是系统的缓存而不是 jvm 的堆内存，所以效率还是蛮高的。

doc values是在字段索引时与倒排索引同时生成。

doc values与倒排索引一样也是基于segment 生成并且是不可变的。

doc1: hello world you and me

doc2: hi, world, how are you

word doc1 doc2

hello \*

world \* \*

you \* \*

and \*

me \*

hi \*

how \*

are \*

hello you --> hello, you

hello --> doc1

you --> doc1,doc2

doc1: hello world you and me

doc2: hi, world, how are you

例如：sort by age，这里是通过 age 排序

doc1: { "name": "jack", "age": 27 }

doc2: { "name": "tom", "age": 30 }

正排索引后的数据格式如：

document name age

doc1 jack 27

doc2 tom 30

### 内核级知识query phase

1、query phase

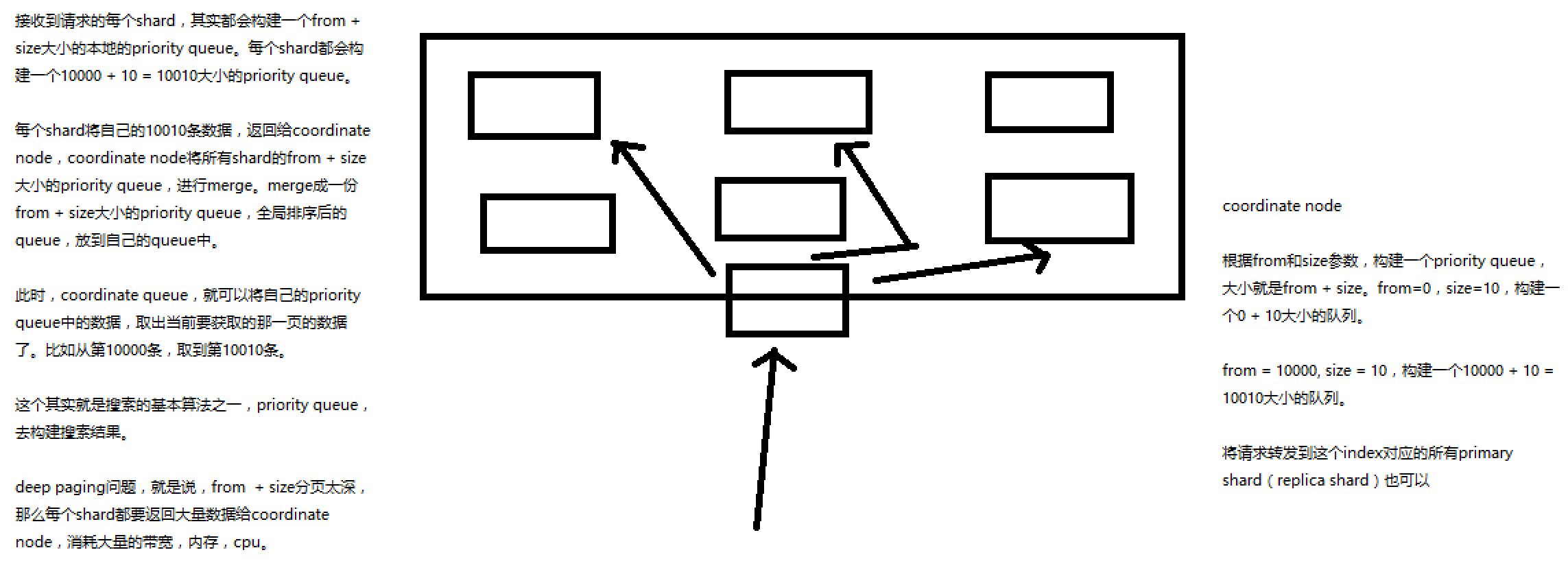
（1）搜索请求发送到某一个coordinate node，构建一个priority queue，长度以paging操作from和size为准，默认为10

（2）coordinate node将请求转发到所有shard，每个shard本地搜索，并构建一个本地的priority queue

（3）各个shard将自己的priority queue返回给coordinate node，并构建一个全局的priority queue

2、replica shard如何提升搜索吞吐量

一次请求要打到所有shard的一个replica/primary上去，如果每个shard都有多个replica，那么同时并发过来的搜索请求可以同时打到其他的replica上去



### 内核级知识fetch phase

1、fetch phbase工作流程

（1）coordinate node构建完priority queue之后，就发送mget请求去所有shard上获取对应的document

（2）各个shard将document返回给coordinate node

（3）coordinate node将合并后的document结果返回给client客户端

2、一般搜索，如果不加from和size，就默认搜索前10条，按照\_score排序

### 搜索相关参数梳理以及 bouncing results 问题解决方案

1、preference：决定了哪些shard会被用来执行搜索操作

可选值：

\_primary, \_primary\_first, \_local, \_only\_node:xyz, \_prefer\_node:xyz, \_shards:2,3

bouncing results问题，两个document排序，field值相同；不同的shard上，可能排序不同；每次请求轮询打到不同的replica shard上；每次页面上看到的搜索结果的排序都不一样。这就是bouncing result，也就是跳跃的结果。

搜索的时候，是轮询将搜索请求发送到每一个replica shard（primary shard），但是在不同的shard上，可能document的排序不同

解决方案就是将preference设置为一个字符串，比如说user\_id，让每个user每次搜索的时候，都使用同一个replica shard去执行，就不会看到bouncing results了

2、timeout：已经讲解过原理了，主要就是限定在一定时间内，将部分获取到的数据直接返回，避免查询耗时过长

3、routing：document文档路由，\_id路由，routing=user\_id，这样的话可以让同一个user对应的数据到一个shard上去

4、search\_type：

可选值：

default：query\_then\_fetch

dfs\_query\_then\_fetch，可以提升revelance sort精准度

### 上机动手实战基于scroll 技术滚动搜索大量数据

如果一次性要查出来比如10万条数据，那么性能会很差，此时一般会采取用scoll滚动查询，一批一批的查，直到所有数据都查询完处理完

使用scoll滚动搜索，可以先搜索一批数据，然后下次再搜索一批数据，以此类推，直到搜索出全部的数据来

scroll搜索会在第一次搜索的时候，保存一个当时的视图快照，之后只会基于该旧的视图快照提供数据搜索，如果这个期间数据变更，是不会让用户看到的

采用基于\_doc进行排序的方式，性能较高

每次发送scroll请求，我们还需要指定一个scoll参数，指定一个时间窗口，每次搜索请求只要在这个时间窗口内能完成就可以了

GET /test\_index/test\_type/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": [ "\_doc" ],

"size": 3

}

结果：

|  |
| --- |
| {  "\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 10,  "max\_score": null,  "hits": [  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "8",  "\_score": null,  "\_source": {  "test\_field": "test client 2"  },  "sort": [  0  ]  },  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "6",  "\_score": null,  "\_source": {  "test\_field": "tes test"  },  "sort": [  0  ]  },  {  "\_index": "test\_index",  "\_type": "test\_type",  "\_id": "AVp4RN0bhjxldOOnBxaE",  "\_score": null,  "\_source": {  "test\_content": "my test"  },  "sort": [  0  ]  }  ]  }  } |

获得的结果会有一个scoll\_id，下一次再发送scoll请求的时候，必须带上这个scoll\_id

GET /\_search/scroll

{

"scroll": "1m",

"scroll\_id" : "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACxeFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAALF8WNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACxhFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAsYhY0b25zVFlWWlRqR3ZJajlfc3BXejJ3"

}

11,4,7

3,2,1

20

scoll，看起来挺像分页的，但是其实使用场景不一样。

>> 分页主要是用来一页一页搜索，给用户看的；

>> scroll主要是用来一批一批检索数据，让系统进行处理的

## 索引管理

### 创建修改以及删除-上机实战

1、为什么我们要手动创建索引？

指定配置信息，以及 mapping 信息。

2、创建索引

创建索引的语法：

PUT /my\_index

{

"settings": { ... any settings ... },

"mappings": {

"type\_one": { ... any mappings ... },

"type\_two": { ... any mappings ... },

...

}

}

创建索引的示例：

PUT /my\_index

{

"settings": {

"number\_of\_shards": 1,

"number\_of\_replicas": 0

},

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "text"

}

}

}

}

}

3、修改索引

PUT /my\_index/\_settings

{

"number\_of\_replicas": 1

}

4、删除索引

DELETE /my\_index

DELETE /index\_one,index\_two

DELETE /index\_\*

DELETE /\_all

elasticsearch.yml

action.destructive\_requires\_name: true

### 修改分词器以及定制自己的分词器-上机实战

1、默认的分词器

默认的分词器是standard分词器。下面是一些标配的分词器的介绍

\*\* standard tokenizer：以单词边界进行切分

\*\* standard token filter：什么都不做

\*\* lowercase token filter：将所有字母转换为小写

\*\* stop token filer（默认被禁用）：移除停用词，比如a the it等等

2、修改分词器的设置

启用english停用词token filter

PUT /my\_index

{

"settings": {

"analysis": {

"analyzer": {

"es\_std": {

"type": "standard",

"stopwords": "\_english\_"

}

}

}

}

}

然后进行分词测试，使用标准的和停用词过滤的分词器进行对比。测试命令如下：

GET /my\_index/\_analyze

{

"analyzer": "standard",

"text": "a dog is in the house"

}

GET /my\_index/\_analyze

{

"analyzer": "es\_std",

"text":"a dog is in the house"

}

3、定制化自己的分词器

PUT /my\_index

{

"settings": {

"analysis": {

"char\_filter": {

"&\_to\_and": {

"type": "mapping",

"mappings": ["&=> and"]

}

},

"filter": {

"my\_stopwords": {

"type": "stop",

"stopwords": ["the", "a"]

}

},

"analyzer": {

"my\_analyzer": {

"type": "custom",

"char\_filter": ["html\_strip", "&\_to\_and"],

"tokenizer": "standard",

"filter": ["lowercase", "my\_stopwords"]

}

}

}

}

}

测试自定义分词器后的效果

GET /my\_index/\_analyze

{

"text": "tom&jerry are a friend in the house, <a>, HAHA!!",

"analyzer": "my\_analyzer"

}

PUT /my\_index/\_mapping/my\_type

{

"properties": {

"content": {

"type": "text",

"analyzer": "my\_analyzer"

}

}

}

### 内核级知识点：type 底层数据结构

type，是一个index中用来区分类似的数据的，类似的数据，但是可能有不同的fields，而且有不同的属性来控制索引建立、分词器

field的value，在底层的lucene中建立索引的时候，全部是**opaque bytes类型**，不区分类型的

lucene是没有type的概念的，在document中，实际上将type作为一个document的field来存储，即\_type，es通过\_type来进行type的过滤和筛选

一个index中的多个type，实际上是放在一起存储的，因此一个index下，不能有多个type重名，而类型或者其他设置不同的，因为那样是无法处理的

如下所示，index 的示例如下。有elactronic\_goods、fresh\_goods两个 type。然后都有 name和 price 字段，不同的是service\_period和 eat\_period字段。

{

"ecommerce": {

"mappings": {

"elactronic\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"service\_period": {

"type": "string"

}

}

},

"fresh\_goods": {

"properties": {

"name": {

"type": "string",

},

"price": {

"type": "double"

},

"eat\_period": {

"type": "string"

}

}

}

}

}

}

elactronic\_goods数据示例：

{

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year"

}

fresh\_goods 数据示例：

{

"name": "aozhou dalongxia",

"price": 199.0,

"eat\_period": "one week"

}

在底层的存储是这样子的。。。。

mapping 样子：

{

"ecommerce": {

"mappings": {

"\_type": {

"type": "string",

"index": "not\_analyzed"

},

"name": {

"type": "string"

}

"price": {

"type": "double"

}

"service\_period": {

"type": "string"

}

"eat\_period": {

"type": "string"

}

}

}

}

elactronic\_goods数据示例：

{

"\_type": "elactronic\_goods",

"name": "geli kongtiao",

"price": 1999.0,

"service\_period": "one year",

"eat\_period": ""

}

fresh\_goods 数据示例：

{

"\_type": "fresh\_goods",

"name": "aozhou dalongxia",

"price": 199.0,

"service\_period": "",

"eat\_period": "one week"

}

最佳实践，将类似结构的type放在一个index下，这些type应该有多个field是相同的

假如说，你将两个type的field完全不同，放在一个index下，那么就每条数据都至少有一半的field在底层的lucene中是空值，会有严重的性能问题

### mapping root object 深入剖析

1、root object

就是某个type对应的mapping json，包括了properties，metadata（\_id，\_source，\_type），settings（analyzer），其他settings（比如include\_in\_all）

命令示意：

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {}

}

}

}

2、properties

properties 的值依次有：type，index，analyzer

命令示意：

PUT /my\_index/\_mapping/my\_type

{

"properties": {

"title": {

"type": "text"

}

}

}

3、\_source

使用\_source的好处如下几点：

（1）查询的时候，直接可以拿到完整的document，不需要先拿document id，再发送一次请求拿document

（2）partial update基于\_source实现

（3）reindex时，直接基于\_source实现，不需要从数据库（或者其他外部存储）查询数据再修改

（4）可以基于\_source定制返回field

（5）debug query更容易，因为可以直接看到\_source

如果不需要上述好处，可以禁用\_source

命令示意：

PUT /my\_index/\_mapping/my\_type2

{

"\_source": {"enabled": false}

}

4、\_all

将所有field打包在一起，作为一个\_all field，建立索引。没指定任何field进行搜索时，就是使用\_all field在搜索。

命令示意：

PUT /my\_index/\_mapping/my\_type3

{

"\_all": {"enabled": false}

}

也可以在field级别设置include\_in\_all field，设置是否要将field的值包含在\_all field中

PUT /my\_index/\_mapping/my\_type4

{

"properties": {

"my\_field": {

"type": "text",

"include\_in\_all": false

}

}

}

5、标识性metadata

\_index，\_type，\_id

### 定制化自己的 dynamic mapping 策略

1、定制dynamic策略

true：遇到陌生字段，就进行dynamic mapping

false：遇到陌生字段，就忽略

strict：遇到陌生字段，就报错

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"title": {

"type": "text"

},

"address": {

"type": "object",

"dynamic": "true"

}

}

}

}

}

PUT /my\_index/my\_type/1

{

"title": "my article",

"content": "this is my article",

"address": {

"province": "guangdong",

"city": "guangzhou"

}

}

{

"error": {

"root\_cause": [

{

"type": "strict\_dynamic\_mapping\_exception",

"reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed"

}

],

"type": "strict\_dynamic\_mapping\_exception",

"reason": "mapping set to strict, dynamic introduction of [content] within [my\_type] is not allowed"

},

"status": 400

}

PUT /my\_index/my\_type/1

{

"title": "my article",

"address": {

"province": "guangdong",

"city": "guangzhou"

}

}

GET /my\_index/\_mapping/my\_type

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic": "strict",

"properties": {

"address": {

"dynamic": "true",

"properties": {

"city": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"province": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

},

"title": {

"type": "text"

}

}

}

}

}

}

2、定制dynamic mapping策略

（1）date\_detection

默认会按照一定格式识别date，比如yyyy-MM-dd。但是如果某个field先过来一个2017-01-01的值，就会被自动dynamic mapping成date，后面如果再来一个"hello world"之类的值，就会报错。可以手动关闭某个type的date\_detection，如果有需要，自己手动指定某个field为date类型。

PUT /my\_index/\_mapping/my\_type

{

"date\_detection": false

}

（2）定制自己的dynamic mapping template（type level）

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{ "en": {

"match": "\*\_en",

"match\_mapping\_type": "string",

"mapping": {

"type": "string",

"analyzer": "english"

}

}}

]

}}}

PUT /my\_index/my\_type/1

{

"title": "this is my first article"

}

PUT /my\_index/my\_type/2

{

"title\_en": "this is my first article"

}

title没有匹配到任何的dynamic模板，默认就是standard分词器，不会过滤停用词，is会进入倒排索引，用is来搜索是可以搜索到的

title\_en匹配到了dynamic模板，就是english分词器，会过滤停用词，is这种停用词就会被过滤掉，用is来搜索就搜索不到了

（3）定制自己的default mapping template（index level）

PUT /my\_index

{

"mappings": {

"\_default\_": {

"\_all": { "enabled": false }

},

"blog": {

"\_all": { "enabled": true }

}

}

}

### 基于 scroll+bulk+索引别名实现零停机重建索引-复杂上机实验

1、重建索引

一个field的设置是不能被修改的，如果要修改一个Field，那么应该重新按照新的mapping，建立一个index，然后将数据批量查询出来，重新用bulk api写入index中

批量查询的时候，建议采用scroll api，并且采用多线程并发的方式来reindex数据，每次scoll就查询指定日期的一段数据，交给一个线程即可

（1）一开始，依靠dynamic mapping，插入数据，但是不小心有些数据是2017-01-01这种日期格式的，所以title这种field被自动映射为了date类型，实际上它应该是string类型的

测试命令：

PUT /my\_index/my\_type/3

{

"title": "2017-01-03"

}

测试结果：

|  |
| --- |
| {  "my\_index": {  "mappings": {  "my\_type": {  "properties": {  "title": {  "type": "date"  }  }  }  }  }  } |

（2）当后期向索引中加入string类型的title值的时候，就会报错

测试命令：

PUT /my\_index/my\_type/4

{

"title": "my first article"

}

测试结果：

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "mapper\_parsing\_exception",  "reason": "failed to parse [title]"  }  ],  "type": "mapper\_parsing\_exception",  "reason": "failed to parse [title]",  "caused\_by": {  "type": "illegal\_argument\_exception",  "reason": "Invalid format: \"my first article\""  }  },  "status": 400  } |

（3）如果此时想修改title的类型，是不可能的

测试命令：

PUT /my\_index/\_mapping/my\_type

{

"properties": {

"title": {

"type": "text"

}

}

}

测试结果：

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "illegal\_argument\_exception",  "reason": "mapper [title] of different type, current\_type [date], merged\_type [text]"  }  ],  "type": "illegal\_argument\_exception",  "reason": "mapper [title] of different type, current\_type [date], merged\_type [text]"  },  "status": 400  } |

（4）此时，唯一的办法，就是进行reindex，也就是说，重新建立一个索引，将旧索引的数据查询出来，再导入新索引

（5）如果说旧索引的名字，是old\_index，新索引的名字是new\_index，终端java应用，已经在使用old\_index在操作了，难道还要去停止java应用，修改使用的index为new\_index，才重新启动java应用吗？这个过程中，就会导致java应用停机，可用性降低

（6）所以说，**给java应用一个别名**，这个别名是指向旧索引的，java应用先用着，java应用先用goods\_index alias来操作，此时实际指向的是旧的my\_index

使用如下命令创建索引别名：

PUT /my\_index/\_alias/goods\_index

（7）新建一个index，调整其title的类型为string

PUT /my\_index\_new

{

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "text"

}

}

}

}

}

（8）使用scroll api将数据批量查询出来

GET /my\_index/\_search?scroll=1m

{

"query": {

"match\_all": {}

},

"sort": ["\_doc"],

"size": 1

}

结果：

|  |
| --- |
| {  "\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAADpAFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6QRY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAOkIWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAADpDFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAA6RBY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",  "took": 1,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": null,  "hits": [  {  "\_index": "my\_index",  "\_type": "my\_type",  "\_id": "2",  "\_score": null,  "\_source": {  "title": "2017-01-02"  },  "sort": [  0  ]  }  ]  }  } |

（9）采用bulk api将scoll查出来的一批数据，批量写入新索引my\_index\_new

POST /\_bulk

{ "index": { "\_index": "my\_index\_new", "\_type": "my\_type", "\_id": "2" }}

{ "title": "2017-01-02" }

（10）反复循环8~9，查询一批又一批的数据出来，采取bulk api将每一批数据批量写入新索引

（11）将goods\_index alias切换到my\_index\_new上去，java应用会直接通过index别名使用新的索引中的数据，java应用程序不需要停机，零提交，高可用

POST /\_aliases

{

"actions": [

{ "remove": { "index": "my\_index", "alias": "goods\_index" }},

{ "add": { "index": "my\_index\_new", "alias": "goods\_index" }}

]

}

（12）直接通过goods\_index别名来查询，是否ok

GET /goods\_index/my\_type/\_search

2、基于alias对client透明切换index

PUT /my\_index\_v1/\_alias/my\_index

client对my\_index进行操作

reindex操作，完成之后，切换v1到v2

POST /\_aliases

{

"actions": [

{ "remove": { "index": "my\_index\_v1", "alias": "my\_index" }},

{ "add": { "index": "my\_index\_v2", "alias": "my\_index" }}

]

}

## 内核原理揭秘

### 倒排索引组成结构以及其索引不可变原因揭秘

倒排索引，是适合用于进行搜索的一个结构数据

倒排索引的结构：

（1）包含这个关键词的document list

（2）包含这个关键词的所有document的数量：IDF（inverse document frequency）

（3）这个关键词在每个document中出现的次数：TF（term frequency）

（4）这个关键词在这个document中的次序

（5）每个document的长度：length norm

（6）包含这个关键词的所有document的平均长度

word doc1 doc2

dog \* \*

hello \*

you \*

倒排索引不可变的**好处**

（1）不需要锁，提升并发能力，避免锁的问题

（2）数据不变，一直保存在os cache中，只要cache内存足够

（3）filter cache一直驻留在内存，因为数据不变

（4）可以压缩，节省cpu和io开销

倒排索引不可变的**坏处**：每次都要重新构建整个索引

### 深度图解剖析Document写入原理

涉及buffer,segment,commit等概念

ES 的大概写入流程（未改造的）：

（1）数据写入buffer

（2）commit point

（3）buffer中的数据写入新的index segment

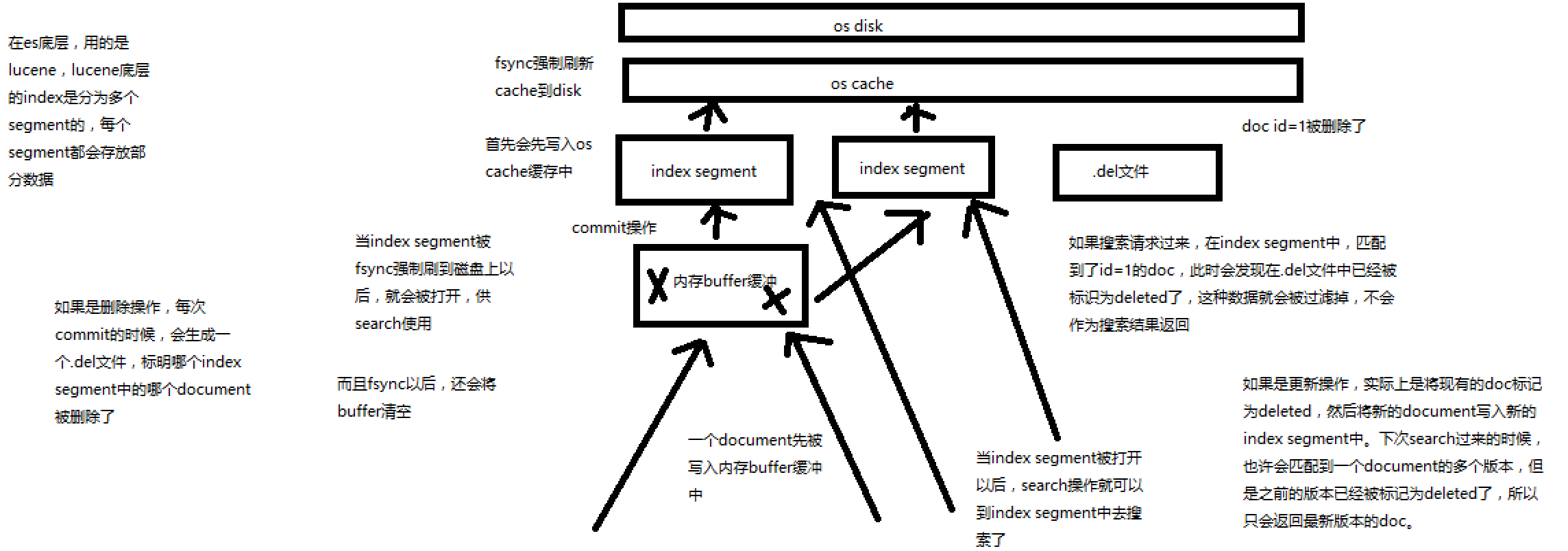
（4）等待在os cache中的index segment被fsync强制刷到磁盘上

（5）新的index sgement被打开，供search使用

（6）buffer被清空

每次commit point时，会有一个.del文件，标记了哪些segment中的哪些document被标记为deleted了

搜索的时候，会依次查询所有的segment，从旧的到新的，比如被修改过的document，在旧的segment中，会标记为deleted，在新的segment中会有其新的数据



### (refresh)优化写入流程实现近 NRT 近实时

涉及的概念有：filesystem cache,refresh

在 Lucene 4.x 版本以前调用 reOpen 操作实现 NRT

在 Lucene 4.x 版本以后可以调用openIfChanged() 实现 NRT操作

SimpleLuceneTests 测试用例里面有 NRT 的例子。

现有流程的问题，每次都必须等待fsync将segment刷入磁盘，才能将segment打开供search使用，这样的话，从一个document写入，到它可以被搜索，可能会超过1分钟！！！这就不是近实时的搜索了！！！主要瓶颈在于fsync实际发生磁盘IO写数据进磁盘，是很耗时的。

所以refresh实现的是文档数据从**内存**到**文件系统缓存**的过程。

refresh 操作有效地对Lucene 的 IndexReader调用了 reopen 操作，使得在那个时间快照进行了更新。这是 Lucene 拥有的近实时搜索 api 的特性。

ES refresh让文档可以搜索到，但是不保证这些信息被写入disk进入一个永久的存储状态，因为它并没有调用fsync，这就不能保证持久性了。让你数据获得持久性的是Lucene commit，这个操作代价比较大。

借助lucene你可以尽可能频繁地调用reopen以使新的文档可以被搜索到，但是你仍然需要调用commit来确保数据写入disk并且fsynced，这样会安全。

写入流程别改进如下：

（1）数据写入buffer

（2）每隔一定时间，buffer中的数据被写入segment文件，但是先写入os cache

（3）只要segment写入os cache，那就直接打开供search使用，不立即执行commit

数据写入os cache，并被打开供搜索的过程，叫做**refresh**，默认是每隔1秒refresh一次。也就是说，每隔一秒就会将buffer中的数据写入一个新的index segment file，先写入os cache中。所以，es是近实时的，数据写入到可以被搜索，默认是1秒。

POST /my\_index/\_refresh，可以手动refresh，一般不需要手动执行，没必要，让es自己搞就可以了

比如说，我们现在的时效性要求，比较低，只要求一条数据写入es，一分钟以后才让我们搜索到就可以了，那么就可以调整refresh interval

调整 refresh interval 的命令是：

PUT /my\_index

{

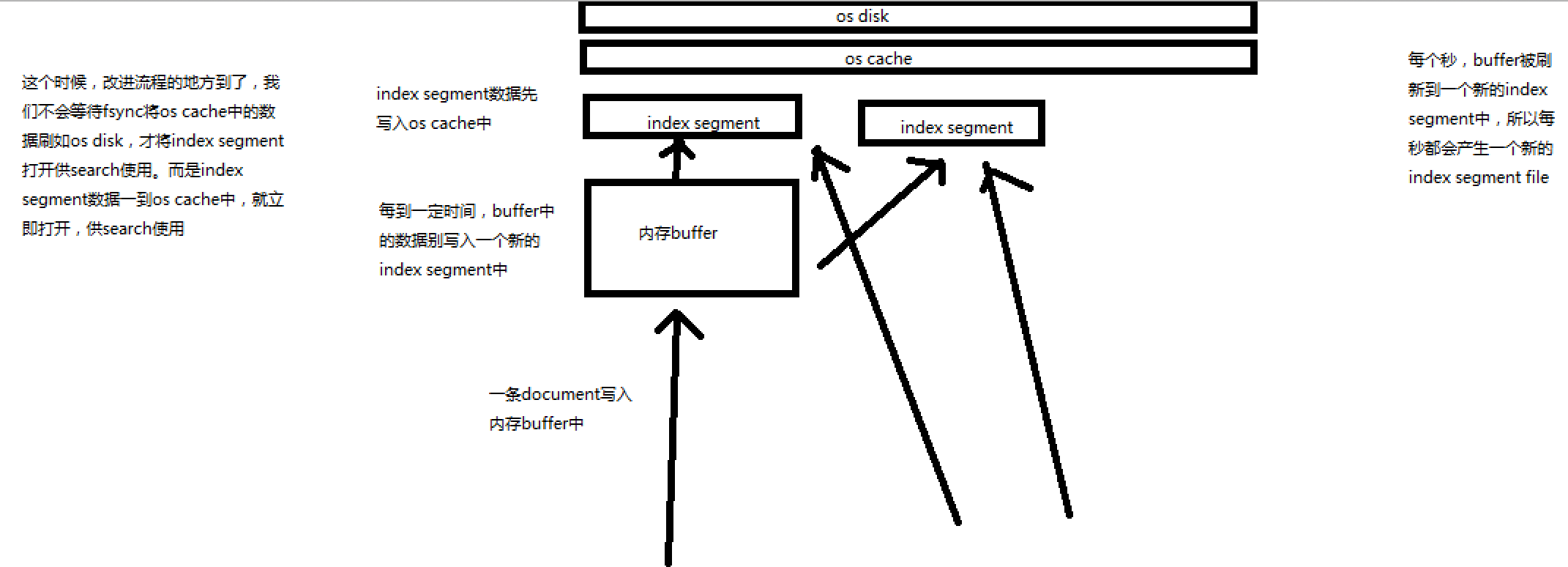
"settings": {

"refresh\_interval": "30s"

}

}

commit。。。稍后就会讲。。。



### (flush)继续优化写入流程实现 durability 可靠存储

设计的概念有：translog,flush

再次优化的写入流程

（1）数据写入buffer缓冲和translog日志文件

（2）每隔一秒钟，buffer中的数据被写入新的segment file，并进入os cache，此时segment被打开并供search使用

（3）buffer被清空

（4）重复1~3，新的segment不断添加，buffer不断被清空，而translog中的数据不断累加

（5）当translog长度达到一定程度的时候，commit操作发生

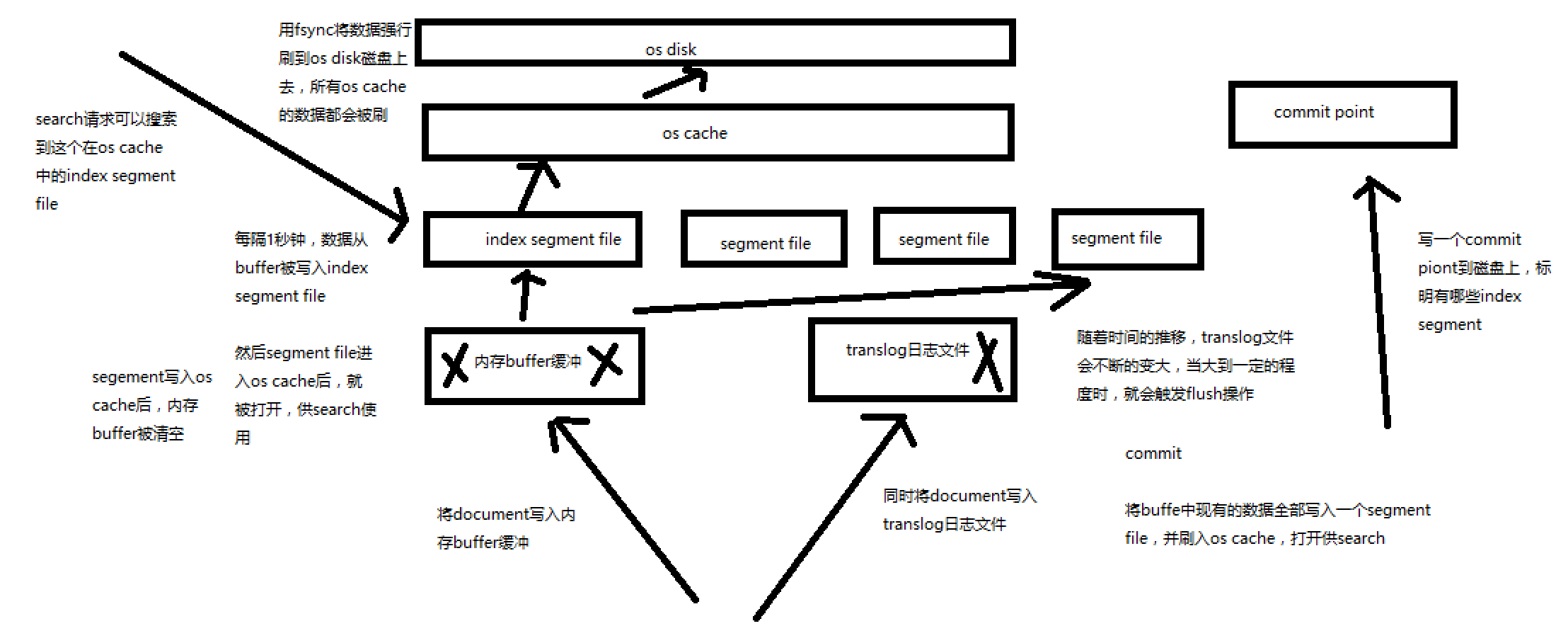
（5-1）buffer中的所有数据写入一个新的segment，并写入os cache，打开供使用

（5-2）buffer被清空

（5-3）一个commit ponit被写入磁盘，标明了所有的index segment

（5-4）filesystem cache中的所有index segment file缓存数据，被fsync强行刷到磁盘上

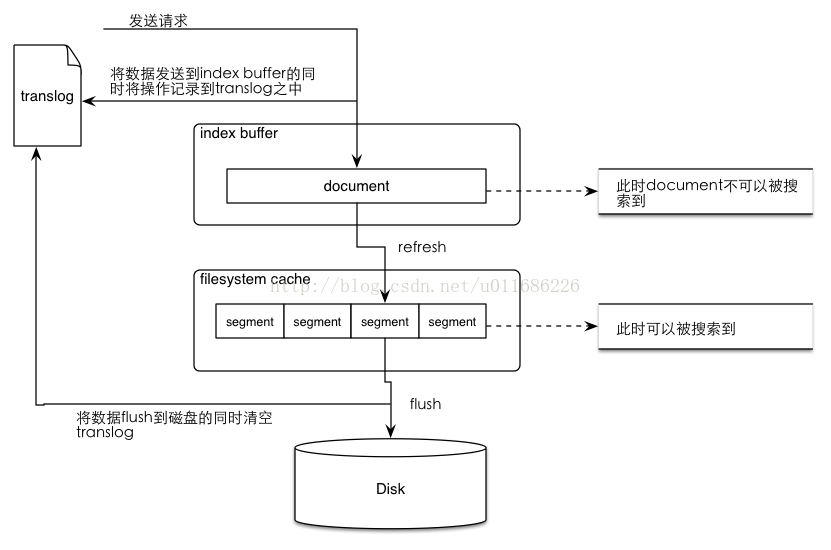
（5-5）现有的translog被清空，创建一个新的translog



ES通过增加了一个在每个shard（一个lucene的索引）上的事务解决这个问题，还未被commit的写操作会被存起来。事务log被fsynced，已经安全了，所以你每时每刻都获得了持久性，甚至对于那些没有被commit的文档，都是这样。因为refresh每秒自动地发生，所以你可以近实时地搜索文档，并且如果有不好的事件发生，事务log可以被替代从而恢复那些丢失的文档。事务log的优越性是它可以被用来做其他的事情，例如提供实时的get\_by\_id。

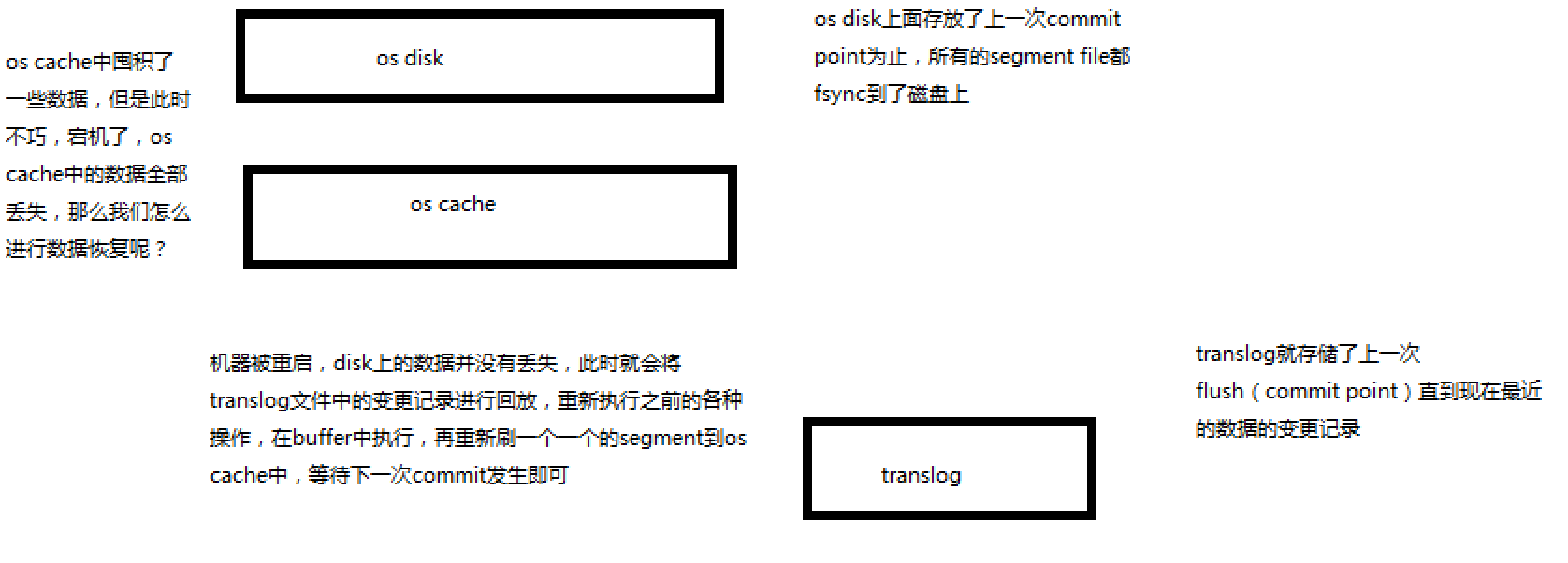
elasticsearch flush高效地触发lucene commit，并同时清空事务log，因为一旦数据在lucene层面提交，持久性将会由lucene保证。Flush同样是一个api，也可以进行微调，虽然通常没有必要这样。Flush自动发生取决于事务log增加了多少操作、它们有多大、最后一次flush何时发生。

当向elasticsearch发送创建document索引请求的时候，document数据会先进入到index buffer之后，与此同时会将操作记录在translog之中，当发生refresh时（数据从index buffer中进入filesystem cache的过程）translog中的操作记录并不会被清除，而是当数据从filesystem cache中被写入磁盘之后才会将translog中清空。而从filesystem cache写入磁盘的过程就是flush。



**问题：基于translog和commit point，如何进行数据恢复**

**答案：**fsync+清空translog，就是flush，默认每隔30分钟flush一次，或者当translog过大的时候，也会flush



一般来说别手动flush，让它自动执行就可以了。手动执行 flush 命令如下：

POST /my\_index/\_flush

translog，每隔5秒被fsync一次到磁盘上。在一次增删改操作之后，当fsync在primary shard和replica shard都成功之后，那次增删改操作才会成功

但是这种在一次增删改时强行fsync translog可能会导致部分操作比较耗时，也可以允许部分数据丢失，设置异步fsync translog

PUT /my\_index/\_settings

{

"index.translog.durability": "async",

"index.translog.sync\_interval": "5s"

}

**总结：**

* **总结一下translog的功能：**

1.保证在filesystem cache中的数据不会因为elasticsearch重启或是发生意外故障的时候丢失。

2.当系统重启时会从translog中恢复之前记录的操作。

3.当对elasticsearch进行CRUD操作的时候，会先到translog之中进行查找，因为tranlog之中保存的是最新的数据。

4.translog的清除时间时进行flush操作之后（将数据从filesystem cache刷入disk之中）。

* **再总结一下flush操作的时间点：**

1.es的各个shard会每个30分钟进行一次flush操作。

2.当translog的数据达到某个上限的时候会进行一次flush操作。

### 最后优化写入流程实现海量磁盘文件合并(segment merge, optimize)

每秒一个segment file，文件过多，而且每次search都要搜索所有的segment，很耗时

默认会在后台执行segment merge操作，在merge的时候，被标记为deleted的document也会被彻底物理删除

每次merge操作的执行流程

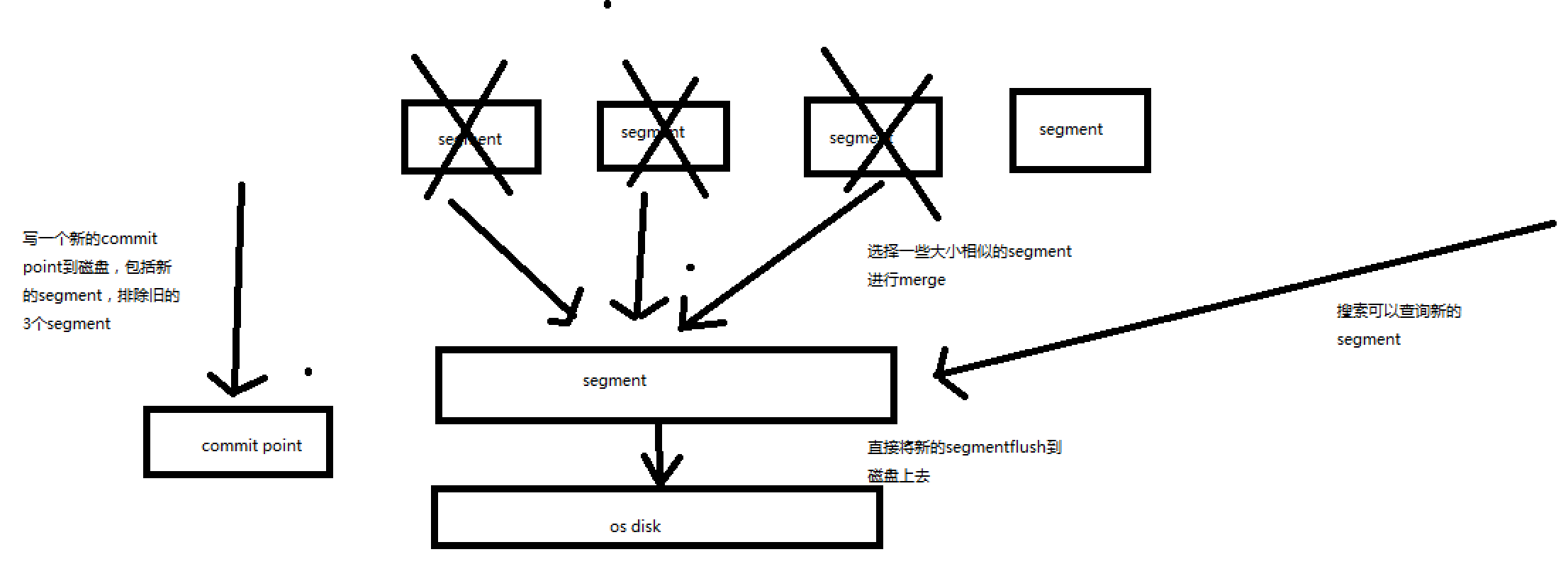
（1）选择一些有相似大小的segment，merge成一个大的segment

（2）将新的segment flush到磁盘上去

（3）写一个新的commit point，包括了新的segment，并且排除旧的那些segment

（4）将新的segment打开供搜索

（5）将旧的segment删除

POST /my\_index/\_optimize?max\_num\_segments=1，尽量不要手动执行，让它自动默认执行就可以了

## JavaAPI 初步使用-员工管理案例

### 基于 Java 实现员工信息的增删改查

强调一下，我们的es讲课的风格

1、es这门技术有点特殊，跟比如其他的像纯java的课程，比如分布式课程，或者大数据类的课程，比如hadoop，spark，storm等。不太一样

2、es非常重要的一个api，是它的restful api，你自己思考一下，掌握这个es的restful api，可以让你执行一些核心的运维管理的操作，比如说创建索引，维护索引，执行各种refresh、flush、optimize操作，查看集群的健康状况，比如还有其他的一些操作，就不在这里枚举了。或者说探查一些数据，可能用java api并不方便。

3、es的学习，首先，你必须学好restful api，然后才是你自己的熟悉语言的api，java api。

这个《核心知识篇（上半季）》，其实主要还是打基础，包括核心的原理，还有核心的操作，还有部分高级的技术和操作，大量的实验，大量的画图，最后初步讲解怎么使用java api

《核心知识篇（下半季）》，包括深度讲解搜索这块技术，还有聚合分析这块技术，包括数据建模，包括java api的复杂使用，有一个项目实战

员工信息

姓名

年龄

职位

国家

入职日期

薪水

我是默认大家至少有java基础的，如果你java一点都不会，请先自己补一下

1、maven依赖

<dependency>

<groupId>org.elasticsearch.client</groupId>

<artifactId>transport</artifactId>

<version>5.2.2</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-api</artifactId>

<version>2.7</version>

</dependency>

<dependency>

<groupId>org.apache.logging.log4j</groupId>

<artifactId>log4j-core</artifactId>

<version>2.7</version>

</dependency>

日志配置文件：log4j2.properties

appender.console.type = Console

appender.console.name = console

appender.console.layout.type = PatternLayout

rootLogger.level = info

rootLogger.appenderRef.console.ref = console

2、构建client

Settings settings = Settings.builder()

.put("cluster.name", "myClusterName").build();

TransportClient client = new PreBuiltTransportClient(settings);

TransportClient client = new PreBuiltTransportClient(Settings.EMPTY)

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host1"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("host2"), 9300));

client.close();

3、创建document

IndexResponse response = client.prepareIndex("index", "type", "1")

.setSource(jsonBuilder()

.startObject()

.field("user", "kimchy")

.field("postDate", new Date())

.field("message", "trying out Elasticsearch")

.endObject()

)

.get();

4、查询document

GetResponse response = client.prepareGet("index", "type", "1").get();

5、修改document

client.prepareUpdate("index", "type", "1")

.setDoc(jsonBuilder()

.startObject()

.field("gender", "male")

.endObject())

.get();

6、删除document

DeleteResponse response = client.prepareDelete("index", "type", "1").get();

主要代码：

|  |
| --- |
| public class EmployeeCRUDApp {  @SuppressWarnings({ "unchecked", "resource" })  public static void main(String[] args) throws Exception {  // 先构建client  Settings settings = Settings.builder()  .put("cluster.name", "elasticsearch")  .build();    TransportClient client = new PreBuiltTransportClient(settings)  .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost"), 9300));    // createEmployee(client);  // getEmployee(client);  // updateEmployee(client);  // deleteEmployee(client);    client.close();  }    /\*\*  \* 创建员工信息（创建一个document）  \* @param client  \*/  private static void createEmployee(TransportClient client) throws Exception {  IndexResponse response = client.prepareIndex("company", "employee", "1")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "jack")  .field("age", 27)  .field("position", "technique")  .field("country", "china")  .field("join\_date", "2017-01-01")  .field("salary", 10000)  .endObject())  .get();  System.out.println(response.getResult());  }    /\*\*  \* 获取员工信息  \* @param client  \* @throws Exception  \*/  private static void getEmployee(TransportClient client) throws Exception {  GetResponse response = client.prepareGet("company", "employee", "1").get();  System.out.println(response.getSourceAsString());  }    /\*\*  \* 修改员工信息  \* @param client  \* @throws Exception  \*/  private static void updateEmployee(TransportClient client) throws Exception {  UpdateResponse response = client.prepareUpdate("company", "employee", "1")  .setDoc(XContentFactory.jsonBuilder()  .startObject()  .field("position", "technique manager")  .endObject())  .get();  System.out.println(response.getResult());  }    /\*\*  \* 删除 员工信息  \* @param client  \* @throws Exception  \*/  private static void deleteEmployee(TransportClient client) throws Exception {  DeleteResponse response = client.prepareDelete("company", "employee", "1").get();  System.out.println(response.getResult());  }  } |

### 基于 Java 对员工信息进行复杂的搜索操作

SearchResponse response = client.prepareSearch("index1", "index2")

.setTypes("type1", "type2")

.setQuery(QueryBuilders.termQuery("multi", "test")) // Query

.setPostFilter(QueryBuilders.rangeQuery("age").from(12).to(18)) // Filter

.setFrom(0).setSize(60)

.get();

需求：

（1）搜索职位中包含technique的员工

（2）同时要求age在30到40岁之间

（3）分页查询，查找第一页

GET /company/employee/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"position": "technique"

}

}

],

"filter": {

"range": {

"age": {

"gte": 30,

"lte": 40

}

}

}

}

},

"from": 0,

"size": 1

}

告诉大家，为什么刚才一边运行创建document，一边搜索什么都没搜索到？？？？

近实时！！！

默认是1秒以后，写入es的数据，才能被搜索到。很明显刚才，写入数据不到一秒，我门就在搜索。

主要代码：

|  |
| --- |
| public class EmployeeSearchApp {  @SuppressWarnings({ "unchecked", "resource" })  public static void main(String[] args) throws Exception {  Settings settings = Settings.builder()  .put("cluster.name", "elasticsearch")  .build();    TransportClient client = new PreBuiltTransportClient(settings)  .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost"), 9300));    // prepareData(client);  executeSearch(client);    client.close();    }  /\*\*  \* 执行搜索操作  \* @param client  \*/  private static void executeSearch(TransportClient client) {  SearchResponse response = client.prepareSearch("company")  .setTypes("employee")  .setQuery(QueryBuilders.matchQuery("position", "technique"))  .setPostFilter(QueryBuilders.rangeQuery("age").from(30).to(40))  .setFrom(0).setSize(1)  .get();    SearchHit[] searchHits = response.getHits().getHits();  for(int i = 0; i < searchHits.length; i++) {  System.out.println(searchHits[i].getSourceAsString());  }  }  /\*\*  \* 准备数据  \* @param client  \*/  private static void prepareData(TransportClient client) throws Exception {  client.prepareIndex("company", "employee", "1")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "jack")  .field("age", 27)  .field("position", "technique software")  .field("country", "china")  .field("join\_date", "2017-01-01")  .field("salary", 10000)  .endObject())  .get();    client.prepareIndex("company", "employee", "2")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "marry")  .field("age", 35)  .field("position", "technique manager")  .field("country", "china")  .field("join\_date", "2017-01-01")  .field("salary", 12000)  .endObject())  .get();    client.prepareIndex("company", "employee", "3")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "tom")  .field("age", 32)  .field("position", "senior technique software")  .field("country", "china")  .field("join\_date", "2016-01-01")  .field("salary", 11000)  .endObject())  .get();    client.prepareIndex("company", "employee", "4")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "jen")  .field("age", 25)  .field("position", "junior finance")  .field("country", "usa")  .field("join\_date", "2016-01-01")  .field("salary", 7000)  .endObject())  .get();    client.prepareIndex("company", "employee", "5")  .setSource(XContentFactory.jsonBuilder()  .startObject()  .field("name", "mike")  .field("age", 37)  .field("position", "finance manager")  .field("country", "usa")  .field("join\_date", "2015-01-01")  .field("salary", 15000)  .endObject())  .get();  }  } |

### 基于 Java 对员工信息进行聚合分析

SearchResponse sr = node.client().prepareSearch()

.addAggregation(

AggregationBuilders.terms("by\_country").field("country")

.subAggregation(AggregationBuilders.dateHistogram("by\_year")

.field("dateOfBirth")

.dateHistogramInterval(DateHistogramInterval.YEAR)

.subAggregation(AggregationBuilders.avg("avg\_children").field("children"))

)

)

.execute().actionGet();

我们先给个需求：

（1）首先按照country国家来进行分组

（2）然后在每个country分组内，再按照入职年限进行分组

（3）最后计算每个分组内的平均薪资

PUT /company

{

"mappings": {

"employee": {

"properties": {

"age": {

"type": "long"

},

"country": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

},

"fielddata": true

},

"join\_date": {

"type": "date"

},

"name": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"position": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

},

"salary": {

"type": "long"

}

}

}

}

}

GET /company/employee/\_search

{

"size": 0,

"aggs": {

"group\_by\_country": {

"terms": {

"field": "country"

},

"aggs": {

"group\_by\_join\_date": {

"date\_histogram": {

"field": "join\_date",

"interval": "year"

},

"aggs": {

"avg\_salary": {

"avg": {

"field": "salary"

}

}

}

}

}

}

}

}

Map<String, Aggregation> aggrMap = searchResponse.getAggregations().asMap();

StringTerms groupByCountry = (StringTerms) aggrMap.get("group\_by\_country");

Iterator<Bucket> groupByCountryBucketIterator = groupByCountry.getBuckets().iterator();

while(groupByCountryBucketIterator.hasNext()) {

Bucket groupByCountryBucket = groupByCountryBucketIterator.next();

System.out.println(groupByCountryBucket.getKey() + "\t" + groupByCountryBucket.getDocCount());

Histogram groupByJoinDate = (Histogram) groupByCountryBucket.getAggregations().asMap().get("group\_by\_join\_date");

Iterator<org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket> groupByJoinDateBucketIterator = groupByJoinDate.getBuckets().iterator();

while(groupByJoinDateBucketIterator.hasNext()) {

org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket groupByJoinDateBucket = groupByJoinDateBucketIterator.next();

System.out.println(groupByJoinDateBucket.getKey() + "\t" + groupByJoinDateBucket.getDocCount());

Avg avgSalary = (Avg) groupByJoinDateBucket.getAggregations().asMap().get("avg\_salary");

System.out.println(avgSalary.getValue());

}

}

client.close();

}

参考代码：

|  |
| --- |
| public class EmployeeAggrApp {  @SuppressWarnings({ "unchecked", "resource" })  public static void main(String[] args) throws Exception {  Settings settings = Settings.builder()  .put("cluster.name", "elasticsearch")  .build();    TransportClient client = new PreBuiltTransportClient(settings)  .addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost"), 9300));    SearchResponse searchResponse = client.prepareSearch("company")  .addAggregation(AggregationBuilders.terms("group\_by\_country").field("country")  .subAggregation(AggregationBuilders  .dateHistogram("group\_by\_join\_date")  .field("join\_date")  .dateHistogramInterval(DateHistogramInterval.YEAR)  .subAggregation(AggregationBuilders.avg("avg\_salary").field("salary")))  )  .execute().actionGet();    Map<String, Aggregation> aggrMap = searchResponse.getAggregations().asMap();    StringTerms groupByCountry = (StringTerms) aggrMap.get("group\_by\_country");  Iterator<Bucket> groupByCountryBucketIterator = groupByCountry.getBuckets().iterator();  while(groupByCountryBucketIterator.hasNext()) {  Bucket groupByCountryBucket = groupByCountryBucketIterator.next();  System.out.println(groupByCountryBucket.getKey() + ":" + groupByCountryBucket.getDocCount());    Histogram groupByJoinDate = (Histogram) groupByCountryBucket.getAggregations().asMap().get("group\_by\_join\_date");  Iterator<org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket> groupByJoinDateBucketIterator = groupByJoinDate.getBuckets().iterator();  while(groupByJoinDateBucketIterator.hasNext()) {  org.elasticsearch.search.aggregations.bucket.histogram.Histogram.Bucket groupByJoinDateBucket = groupByJoinDateBucketIterator.next();  System.out.println(groupByJoinDateBucket.getKey() + ":" +groupByJoinDateBucket.getDocCount());    Avg avg = (Avg) groupByJoinDateBucket.getAggregations().asMap().get("avg\_salary");  System.out.println(avg.getValue());  }  }    client.close();  }    } |

# 高手进阶篇

## 结构化搜索-案例

### IT 技术论坛案例背景介绍

IT技术论坛，案例背景

IT技术论坛中相关的数据，会在es中建立数据的索引

深度讲解搜索，数据分析，数据建模

特色：纯手工画图剖析各种原理，纯实战驱动讲解各种知识点，知识体系的细致和完整

怎么实战驱动？

核心知识篇，上半季，我们也是纯实战驱动，但是之前没有一个统一的案例背景

IT技术论坛，发各种IT技术的帖子：一种是自己研究了一个技术，就发出来一些研究心得；自己遇到了问题，发个帖子问一问。帖子会有人回复，还会有人浏览。当然了，还有一些论坛会提供简单社交的一个功能，互相加好友，互相关注，互相点赞，之类的。

在IT技术论坛的背景下，去开发一些跟案例背景相关的搜索或者数据分析，或者数据建模的需求，用每一讲学到的知识点，去接解决一些问题

既可以学到知识和技术，也可以在真实的案例背景中练习一下学到的东西

### 案例- term filter 来搜索数据

1、根据用户ID、是否隐藏、帖子ID、发帖日期来搜索帖子

（1）插入一些测试帖子数据

POST /forum/article/\_bulk

{ "index": { "\_id": 1 }}

{ "articleID" : "XHDK-A-1293-#fJ3", "userID" : 1, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 2 }}

{ "articleID" : "KDKE-B-9947-#kL5", "userID" : 1, "hidden": false, "postDate": "2017-01-02" }

{ "index": { "\_id": 3 }}

{ "articleID" : "JODL-X-1937-#pV7", "userID" : 2, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 4 }}

{ "articleID" : "QQPX-R-3956-#aD8", "userID" : 2, "hidden": true, "postDate": "2017-01-02" }

初步来说，就先搞4个字段，因为整个es是支持json document格式的，所以说扩展性和灵活性非常之好。如果后续随着业务需求的增加，要在document中增加更多的field，那么我们可以很方便的随时添加field。但是如果是在关系型数据库中，比如mysql，我们建立了一个表，现在要给表中新增一些column，那就很坑爹了，必须用复杂的修改表结构的语法去执行。而且可能对系统代码还有一定的影响。

GET /forum/\_mapping/article

结果：

|  |
| --- |
| {  "forum": {  "mappings": {  "article": {  "properties": {  "articleID": {  "type": "text",  "fields": {  "keyword": {  "type": "keyword",  "ignore\_above": 256  }  }  },  "hidden": {  "type": "boolean"  },  "postDate": {  "type": "date"  },  "userID": {  "type": "long"  }  }  }  }  }  } |

现在es 5.2版本，type=text，默认会设置两个field，一个是field本身，比如articleID，就是分词的；还有一个的话，就是field.keyword，articleID.keyword，默认不分词，会最多保留256个字符

应该是两种类型，一种是 keyword，不分词的。一种是 text，分词的建立索引。

（2）根据用户ID搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"userID" : 1

}

}

}

}

}

term filter/query：对搜索文本不分词，直接拿去倒排索引中匹配，你输入的是什么，就去匹配什么

比如说，如果对搜索文本进行分词的话，“helle world” --> “hello”和“world”，两个词分别去倒排索引中匹配

term，“hello world” --> “hello world”，直接去倒排索引中匹配“hello world”

（3）搜索没有隐藏的帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"hidden" : false

}

}

}

}

}

（4）根据发帖日期搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"postDate" : "2017-01-01"

}

}

}

}

}

（5）根据帖子ID搜索帖子

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 1,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 0,  "max\_score": null,  "hits": []  }  } |

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID.keyword" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 1,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01"  }  }  ]  }  } |

articleID.keyword，是es最新版本内置建立的field，就是不分词的。所以一个articleID过来的时候，会建立两次索引，一次是自己本身，是要分词的，分词后放入倒排索引；另外一次是基于articleID.keyword，不分词，保留256个字符最多，直接一个字符串放入倒排索引中。

所以term filter，对text过滤，可以考虑使用内置的field.keyword来进行匹配。但是有个问题，默认就保留256个字符。所以尽可能还是自己去手动建立索引，指定not\_analyzed吧。在最新版本的es中，不需要指定not\_analyzed也可以，将type=keyword即可。

（6）查看分词

GET /forum/\_analyze

{

"field": "articleID",

"text": "XHDK-A-1293-#fJ3"

}

默认是analyzed的text类型的field，建立倒排索引的时候，就会对所有的articleID分词，分词以后，原本的articleID就没有了，只有分词后的各个word存在于倒排索引中。

term，是不对搜索文本分词的，XHDK-A-1293-#fJ3 --> XHDK-A-1293-#fJ3；但是articleID建立索引的时候，XHDK-A-1293-#fJ3 --> xhdk，a，1293，fj3

（7）重建索引

线删除索引

DELETE /forum

然后重建索引

PUT /forum

{

"mappings": {

"article": {

"properties": {

"articleID": {

"type": "keyword"

}

}

}

}

}

添加

POST /forum/article/\_bulk

{ "index": { "\_id": 1 }}

{ "articleID" : "XHDK-A-1293-#fJ3", "userID" : 1, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 2 }}

{ "articleID" : "KDKE-B-9947-#kL5", "userID" : 1, "hidden": false, "postDate": "2017-01-02" }

{ "index": { "\_id": 3 }}

{ "articleID" : "JODL-X-1937-#pV7", "userID" : 2, "hidden": false, "postDate": "2017-01-01" }

{ "index": { "\_id": 4 }}

{ "articleID" : "QQPX-R-3956-#aD8", "userID" : 2, "hidden": true, "postDate": "2017-01-02" }

（8）重新根据帖子ID和发帖日期进行搜索

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"term" : {

"articleID" : "XHDK-A-1293-#fJ3"

}

}

}

}

}

2、梳理学到的知识点

（1）term filter：根据exact value进行搜索，数字、boolean、date天然支持

（2）text需要建索引时指定为not\_analyzed，才能用term query

（3）相当于SQL中的单个where条件

select \*

from forum.article

where articleID='XHDK-A-1293-#fJ3'

### 原理-filter 执行原理深度剖析(bitset 机制与 caching 机制)

课程大纲

（1）在倒排索引中查找搜索串，获取document list

date来举例

word doc1 doc2 doc3

2017-01-01 \* \*

2017-02-02 \* \*

2017-03-03 \* \* \*

filter：2017-02-02

到倒排索引中一找，发现2017-02-02对应的document list是doc2,doc3

（2）为每个在倒排索引中搜索到的结果，构建一个bitset，[0, 0, 0, 1, 0, 1]

非常重要

使用找到的doc list，构建一个bitset，就是一个二进制的数组，数组每个元素都是0或1，用来标识一个doc对一个filter条件是否匹配，如果匹配就是1，不匹配就是0

[0, 1, 1]

doc1：不匹配这个filter的

doc2和do3：是匹配这个filter的

尽可能用简单的数据结构去实现复杂的功能，可以节省内存空间，提升性能

（3）遍历每个过滤条件对应的bitset，优先从最稀疏的开始搜索，查找满足所有条件的document

后面会讲解，一次性其实可以在一个search请求中，发出多个filter条件，每个filter条件都会对应一个bitset

遍历每个filter条件对应的bitset，先从最稀疏的开始遍历

[0, 0, 0, 1, 0, 0]：比较稀疏

[0, 1, 0, 1, 0, 1]

先遍历比较稀疏的bitset，就可以先过滤掉尽可能多的数据

遍历所有的bitset，找到匹配所有filter条件的doc

请求：filter，postDate=2017-01-01，userID=1

postDate: [0, 0, 1, 1, 0, 0]

userID: [0, 1, 0, 1, 0, 1]

遍历完两个bitset之后，找到的匹配所有条件的doc，就是doc4

就可以将document作为结果返回给client了

（4）caching bitset，跟踪query，在最近256个query中超过一定次数的过滤条件，缓存其bitset。对于小segment（<1000，或<3%），不缓存bitset。

比如postDate=2017-01-01，[0, 0, 1, 1, 0, 0]，可以缓存在内存中，这样下次如果再有这个条件过来的时候，就不用重新扫描倒排索引，反复生成bitset，可以大幅度提升性能。

在最近的256个filter中，有某个filter超过了一定的次数，次数不固定，就会自动缓存这个filter对应的bitset

segment（上半季），filter针对小segment获取到的结果，可以不缓存，segment记录数<1000，或者segment大小<index总大小的3%

segment数据量很小，此时哪怕是扫描也很快；segment会在后台自动合并，小segment很快就会跟其他小segment合并成大segment，此时就缓存也没有什么意义，segment很快就消失了

针对一个小segment的bitset，[0, 0, 1, 0]

filter比query的好处就在于会caching，但是之前不知道caching的是什么东西，实际上并不是一个filter返回的完整的doc list数据结果。而是filter bitset缓存起来。下次不用扫描倒排索引了。

（5）filter大部分情况下来说，在query之前执行，先尽量过滤掉尽可能多的数据

query：是会计算doc对搜索条件的relevance score，还会根据这个score去排序

filter：只是简单过滤出想要的数据，不计算relevance score，也不排序

（6）如果document有新增或修改，那么cached bitset会被自动更新

postDate=2017-01-01，[0, 0, 1, 0]

document，id=5，postDate=2017-01-01，会自动更新到postDate=2017-01-01这个filter的bitset中，全自动，缓存会自动更新。postDate=2017-01-01的bitset，[0, 0, 1, 0, 1]

document，id=1，postDate=2016-12-30，修改为postDate-2017-01-01，此时也会自动更新bitset，[1, 0, 1, 0, 1]

（7）以后只要是有相同的filter条件的，会直接来使用这个过滤条件对应的cached bitset

### 案例- bool 组合多个 filter 条件来搜索数据

1、搜索发帖日期为2017-01-01，或者帖子ID为XHDK-A-1293-#fJ3的帖子，同时要求帖子的发帖日期绝对不为2017-01-02

等价的 SQL 是：

select \*

from forum.article

where (post\_date='2017-01-01' or article\_id='XHDK-A-1293-#fJ3')

and post\_date!='2017-01-02'

ES 中查询命令是：

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{"term": { "postDate": "2017-01-01" }},

{"term": { "articleID": "XHDK-A-1293-#fJ3"}}

],

"must\_not": {

"term": {

"postDate": "2017-01-02"

}

}

}

}

}

}

}

must，should，must\_not，filter：必须匹配，可以匹配其中任意一个即可，必须不匹配

2、搜索帖子ID为XHDK-A-1293-#fJ3，或者：帖子ID为JODL-X-1937-#pV7而且发帖日期为2017-01-01的帖子

select \*

from forum.article

where article\_id='XHDK-A-1293-#fJ3'

or (article\_id='JODL-X-1937-#pV7' and post\_date='2017-01-01')

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"should": [

{

"term": {

"articleID": "XHDK-A-1293-#fJ3"

}

},

{

"bool": {

"must": [

{

"term":{

"articleID": "JODL-X-1937-#pV7"

}

},

{

"term": {

"postDate": "2017-01-01"

}

}

]

}

}

]

}

}

}

}

}

3、梳理学到的知识点

（1）bool：must，must\_not，should，组合多个过滤条件

（2）bool可以嵌套

（3）相当于SQL中的多个and条件：当你把搜索语法学好了以后，基本可以实现部分常用的sql语法对应的功能

### 案例- terms 搜索多个值以及多值搜索结果优化

语法格式如下：

term: {"field": "value"}

terms: {"field": ["value1", "value2"]}

相当于sql中的in

select \* from tbl where col in ("value1", "value2")

1、为帖子数据增加tag字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"tag" : ["java", "hadoop"]} }

{ "update": { "\_id": "2"} }

{ "doc" : {"tag" : ["java"]} }

{ "update": { "\_id": "3"} }

{ "doc" : {"tag" : ["hadoop"]} }

{ "update": { "\_id": "4"} }

{ "doc" : {"tag" : ["java", "elasticsearch"]} }

2、搜索articleID为KDKE-B-9947-#kL5或QQPX-R-3956-#aD8的帖子，搜索tag中包含java的帖子

搜索articleID为KDKE-B-9947-#kL5或QQPX-R-3956-#aD8的帖子：

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"terms": {

"articleID": [

"KDKE-B-9947-#kL5",

"QQPX-R-3956-#aD8"

]

}

}

}

}

}

搜索tag中包含java的帖子：

GET /forum/article/\_search

{

"query" : {

"constant\_score" : {

"filter" : {

"terms" : {

"tag" : ["java"]

}

}

}

}

}

结果：

|  |
| --- |
| "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 1,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ]  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "4",  "\_score": 1,  "\_source": {  "articleID": "QQPX-R-3956-#aD8",  "userID": 2,  "hidden": true,  "postDate": "2017-01-02",  "tag": [  "java",  "elasticsearch"  ]  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 1,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ]  }  }  ]  }  } |

3、优化搜索结果，仅仅搜索tag只包含java的帖子

增加 tag\_cnd字段。手动添加索引数据：

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"tag\_cnt" : 2} }

{ "update": { "\_id": "2"} }

{ "doc" : {"tag\_cnt" : 1} }

{ "update": { "\_id": "3"} }

{ "doc" : {"tag\_cnt" : 1} }

{ "update": { "\_id": "4"} }

{ "doc" : {"tag\_cnt" : 2} }

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"bool": {

"must": [

{

"term": {

"tag\_cnt": 1

}

},

{

"terms": {

"tag": ["java"]

}

}

]

}

}

}

}

}

["java", "hadoop", "elasticsearch"]

4、学到的知识点梳理

（1）terms多值搜索

（2）优化terms多值搜索的结果

（3）相当于SQL中的in语句

### 案例- range filter 来进行范围过滤

1、为帖子数据增加浏览量的字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"view\_cnt" : 30} }

{ "update": { "\_id": "2"} }

{ "doc" : {"view\_cnt" : 50} }

{ "update": { "\_id": "3"} }

{ "doc" : {"view\_cnt" : 100} }

{ "update": { "\_id": "4"} }

{ "doc" : {"view\_cnt" : 80} }

2、搜索浏览量在30~60之间的帖子

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"view\_cnt": {

"gt": 30,

"lt": 60

}

}

}

}

}

}

gte

lte

3、搜索发帖日期在最近1个月的帖子

先添加一条数据

POST /forum/article/\_bulk

{ "index": { "\_id": 5 }}

{ "articleID": "DHJK-B-1395-#Ky5", "userID": 3, "hidden": false, "postDate": "2017-03-01", "tag": ["elasticsearch"], "tag\_cnt": 1, "view\_cnt": 10 }

命令：使用 ||-30d 的语法，就是时间的 30 天以前。now-30d

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"postDate": {

"gt": "2017-03-10||-30d"

}

}

}

}

}

}

GET /forum/article/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"postDate": {

"gt": "now-30d"

}

}

}

}

}

}

4、梳理一下学到的知识点

（1）range，sql中的between，或者是>=1，<=1

（2）range做范围过滤

## 深度解密搜索技术-高级搜索案例

### 案例-控制全文检索结果的精确度

1、为帖子数据增加标题字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"title" : "this is java and elasticsearch blog"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"title" : "this is java blog"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"title" : "this is elasticsearch blog"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"title" : "this is java, elasticsearch, hadoop blog"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"title" : "this is spark blog"} }

2、搜索标题中包含java或elasticsearch的blog

这个，就跟之前的那个term query，不一样了。不是搜索exact value，是进行full text全文检索。

match query，是负责进行**全文检索**的。当然，如果要检索的field，是not\_analyzed类型的，那么match query也相当于term query。

命令如下：

GET /forum/article/\_search

{

"query": {

"match": {

"title": "java elasticsearch"

}

}

}

3、搜索标题中包含java和elasticsearch的blog

搜索结果精准控制的第一步：灵活使用and关键字，如果你是希望所有的搜索关键字都要匹配的，那么就用and，可以实现单纯match query无法实现的效果

命令如下：

GET /forum/article/\_search

{

"query": {

"match": {

"title": {

"query": "java elasticsearch",

"operator": "and"

}

}

}

}

4、搜索包含java，elasticsearch，spark，hadoop，4个关键字中，至少3个的blog

控制搜索结果的精准度的第二步：指定一些关键字中，必须至少匹配其中的多少个关键字，才能作为结果返回

命令如下：

GET /forum/article/\_search

{

"query": {

"match": {

"title": {

"query": "java elasticsearch spark hadoop",

"minimum\_should\_match": "75%"

}

}

}

}

5、用bool组合多个搜索条件，来搜索title

使用 bool 也可以达到多个词的组合搜索。

GET /forum/article/\_search

{

"query": {

"bool": {

"must": { "match": { "title": "java" }},

"must\_not": { "match": { "title": "spark" }},

"should": [

{ "match": { "title": "hadoop" }},

{ "match": { "title": "elasticsearch" }}

]

}

}

}

6、bool组合多个搜索条件，如何计算relevance score

**must和should搜索对应的分数，加起来，除以must和should的总数**

排名第一：java，同时包含should中所有的关键字，hadoop，elasticsearch

排名第二：java，同时包含should中的elasticsearch

排名第三：java，不包含should中的任何关键字

should是可以影响相关度分数的

must是确保说，谁必须有这个关键字，同时会根据这个must的条件去计算出document对这个搜索条件的relevance score

在满足must的基础之上，should中的条件，不匹配也可以，但是如果匹配的更多，那么document的relevance score就会更高

结果：

|  |
| --- |
| {  "took": 6,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1.3375794,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "4",  "\_score": 1.3375794,  "\_source": {  "articleID": "QQPX-R-3956-#aD8",  "userID": 2,  "hidden": true,  "postDate": "2017-01-02",  "tag": [  "java",  "elasticsearch"  ],  "tag\_cnt": 2,  "view\_cnt": 80,  "title": "this is java, elasticsearch, hadoop blog"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 0.53484553,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ],  "tag\_cnt": 2,  "view\_cnt": 30,  "title": "this is java and elasticsearch blog"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 0.19856805,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog"  }  }  ]  }  } |

7、搜索java，hadoop，spark，elasticsearch，至少包含其中3个关键字

默认情况下，should是可以不匹配任何一个的，比如上面的搜索中，this is java blog，就不匹配任何一个should条件

但是有个例外的情况，如果没有must的话，那么should中必须至少匹配一个才可以

比如下面的搜索，should中有4个条件，默认情况下，只要满足其中一个条件，就可以匹配作为结果返回

但是可以精准控制，should的4个条件中，至少匹配几个才能作为结果返回

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "java" }},

{ "match": { "title": "elasticsearch" }},

{ "match": { "title": "hadoop" }},

{ "match": { "title": "spark" }}

],

"minimum\_should\_match": 3

}

}

}

梳理一下学到的知识点

1、全文检索的时候，进行多个值的检索，有两种做法，match query；should

2、控制搜索结果精准度：and operator，minimum\_should\_match

### 原理-基于 term+bool 实现的 multiword 搜索底层原理剖析

1、普通match如何转换为term+should

{

"match": { "title": "java elasticsearch"}

}

使用诸如上面的match query进行多值搜索的时候，es会在底层自动将这个match query转换为bool的语法

bool should，指定多个搜索词，同时使用term query

{

"bool": {

"should": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }}

]

}

}

2、and match如何转换为term+must

{

"match": {

"title": {

"query": "java elasticsearch",

"operator": "and"

}

}

}

{

"bool": {

"must": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }}

]

}

}

3、minimum\_should\_match如何转换

{

"match": {

"title": {

"query": "java elasticsearch hadoop spark",

"minimum\_should\_match": "75%"

}

}

}

{

"bool": {

"should": [

{ "term": { "title": "java" }},

{ "term": { "title": "elasticsearch" }},

{ "term": { "title": "hadoop" }},

{ "term": { "title": "spark" }}

],

"minimum\_should\_match": 3

}

}

上一讲，为啥要讲解两种实现multi-value搜索的方式呢？实际上，就是给这一讲进行铺垫的。match query --> bool + term。

### boost 的细粒度搜索条件控制权重

需求：搜索标题中包含java的帖子，同时呢，如果标题中包含hadoop或elasticsearch就优先搜索出来，同时呢，如果一个帖子包含java hadoop，一个帖子包含java elasticsearch，包含hadoop的帖子要比elasticsearch优先搜索出来

知识点，搜索条件的权重，boost，可以将某个搜索条件的权重加大，此时当匹配这个搜索条件和匹配另一个搜索条件的document，计算relevance score时，匹配权重更大的搜索条件的document，relevance score会更高，当然也就会优先被返回回来

默认情况下，搜索条件的权重都是一样的，都是1

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "blog"

}

}

],

"should": [

{

"match": {

"title": {

"query": "java"

}

}

},

{

"match": {

"title": {

"query": "hadoop"

}

}

},

{

"match": {

"title": {

"query": "elasticsearch"

}

}

},

{

"match": {

"title": {

"query": "spark",

"boost": 5

}

}

}

]

}

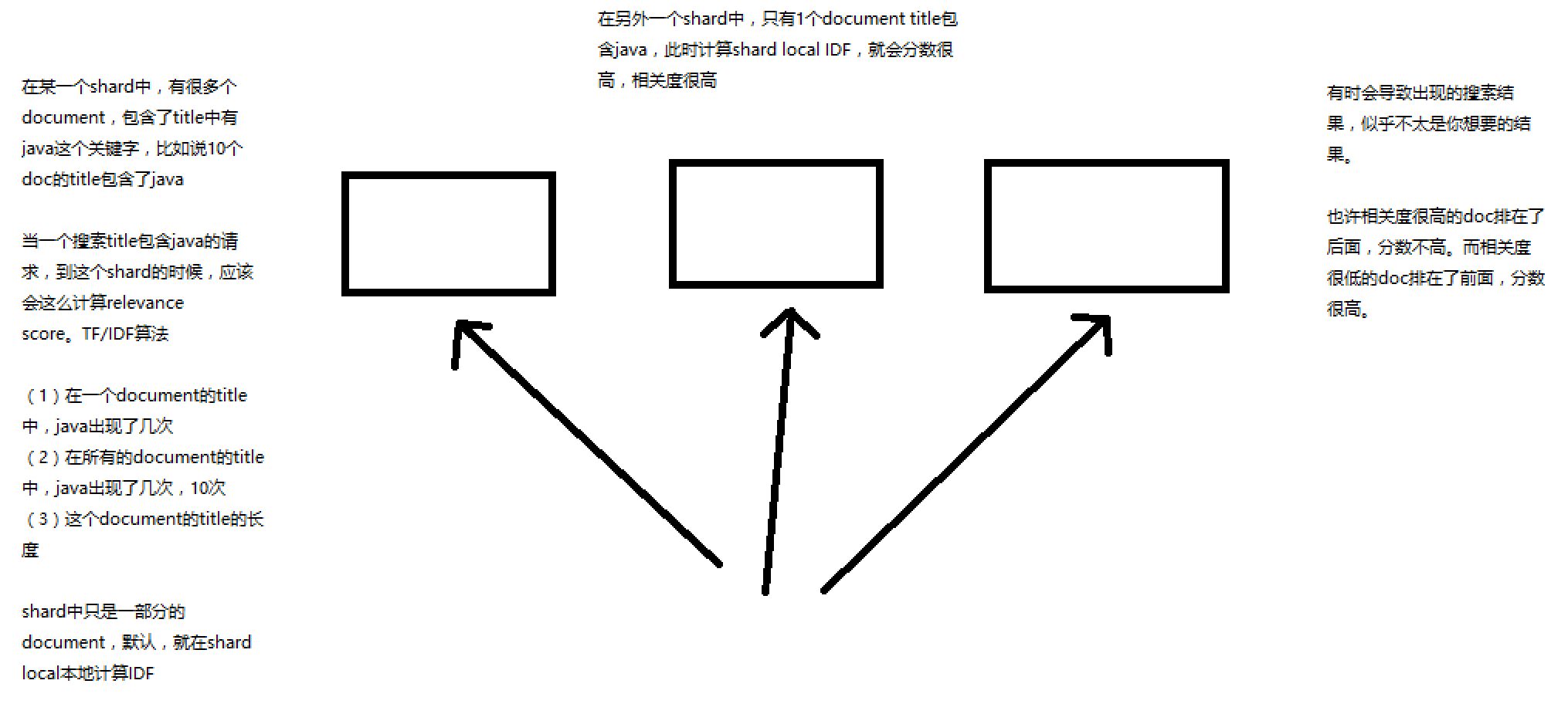
}

}

### 原理-多 shard 场景下 relevance score 不准确问题大揭秘

1、多shard场景下relevance score不准确问题大揭秘

如果你的一个index有多个shard的话，可能搜索结果会不准确

如下进行画图讲解：

2、如何解决该问题？

（1）生产环境下：数据量大，尽可能实现均匀分配

数据量很大的话，其实一般情况下，在概率学的背景下，es都是在多个shard中均匀路由数据的，路由的时候根据\_id，负载均衡

比如说有10个document，title都包含java，一共有5个shard，那么在概率学的背景下，如果负载均衡的话，其实每个shard都应该有2个doc，title包含java

如果说数据分布均匀的话，其实就没有刚才说的那个问题了

（2）测试环境下：将索引的primary shard设置为1个，number\_of\_shards=1，index settings

如果说只有一个shard，那么当然，所有的document都在这个shard里面，就没有这个问题了

（3）测试环境下：搜索附带search\_type=dfs\_query\_then\_fetch参数，会将local IDF取出来计算global IDF

计算一个doc的相关度分数的时候，就会将所有shard对的local IDF计算一下，获取出来，在本地进行global IDF分数的计算，会将所有shard的doc作为上下文来进行计算，也能确保准确性。但是production生产环境下，不推荐这个参数，因为性能很差。

### 案例- dis\_max实现 best fields 策略进行多字段搜索

1、为帖子数据增加content字段

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"content" : "i like to write best elasticsearch article"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"content" : "i think java is the best programming language"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"content" : "i am only an elasticsearch beginner"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"content" : "elasticsearch and hadoop are all very good solution, i am a beginner"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"content" : "spark is best big data solution based on scala ,an programming language similar to java"} }

2、搜索title或content中包含java或solution的帖子

下面这个就是multi-field搜索，多字段搜索

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{ "match": { "title": "java solution" }},

{ "match": { "content": "java solution" }}

]

}

}

}

3、结果分析

疑问：期望的是doc5，结果是doc2,doc4排在了前面，？？？

**解答：**计算每个document的relevance score：每个query的分数，乘以matched query数量，除以总query数量

>> 算一下doc4的分数

{ "match": { "title": "java solution" }}，针对doc4，是有一个分数的

{ "match": { "content": "java solution" }}，针对doc4，也是有一个分数的

所以是两个分数加起来，比如说，1.1 + 1.2 = 2.3

matched query数量 = 2

总query数量 = 2

2.3 \* 2 / 2 = 2.3

>> 算一下doc5的分数

{ "match": { "title": "java solution" }}，针对doc5，是没有分数的

{ "match": { "content": "java solution" }}，针对doc5，是有一个分数的

所以说，只有一个query是有分数的，比如2.3

matched query数量 = 1

总query数量 = 2

2.3 \* 1 / 2 = 1.15

doc5的分数 = 1.15 < doc4的分数 = 2.3

4、best fields策略，dis\_max

best fields策略，就是说，搜索到的结果，应该是某一个field中匹配到了尽可能多的关键词，被排在前面；而不是尽可能多的field匹配到了少数的关键词，排在了前面

**dis\_max语法，直接取多个query中，分数最高的那一个query的分数即可**

{ "match": { "title": "java solution" }}，针对doc4，是有一个分数的，1.1

{ "match": { "content": "java solution" }}，针对doc4，也是有一个分数的，1.2

取最大分数，1.2

{ "match": { "title": "java solution" }}，针对doc5，是没有分数的

{ "match": { "content": "java solution" }}，针对doc5，是有一个分数的，2.3

取最大分数，2.3

然后doc4的分数 = 1.2 < doc5的分数 = 2.3，所以doc5就可以排在更前面的地方，符合我们的需要

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java solution" }},

{ "match": { "content": "java solution" }}

]

}

}

}

剩下的还有 tie\_breaker参数，在下一接中讲解：

### 案例- dis\_max参数tie\_breaker优化搜索效果

1、搜索title或content中包含java beginner的帖子

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java beginner" }},

{ "match": { "body": "java beginner" }}

]

}

}

}

有些场景不是太好复现的，因为是这样，你需要尝试去构造不同的文本，然后去构造一些搜索出来，去达到你要的一个效果

可能在实际场景中出现的一个情况是这样的：

（1）某个帖子，doc1，title中包含java，content不包含java beginner任何一个关键词

（2）某个帖子，doc2，content中包含beginner，title中不包含任何一个关键词

（3）某个帖子，doc3，title中包含java，content中包含beginner

（4）最终搜索，可能出来的结果是，doc1和doc2排在doc3的前面，而不是我们期望的doc3排在最前面

dis\_max，只是取分数最高的那个query的分数而已。

2、dis\_max只取某一个query最大的分数，完全不考虑其他query的分数

3、使用tie\_breaker将其他query的分数也考虑进去

tie\_breaker参数的意义，在于说，将其他query的分数，乘以tie\_breaker，然后综合与最高分数的那个query的分数，综合在一起进行计算，除了取最高分以外，还会考虑其他的query的分数

tie\_breaker的值，在0~1之间，是个小数，就ok

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{ "match": { "title": "java beginner" }},

{ "match": { "body": "java beginner" }}

],

"tie\_breaker": 0.3

}

}

}

### 案例-multi\_match语法实现dis\_max+tie\_breaker

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "java solution",

"type": "best\_fields",

"fields": [ "title^2", "content" ],

"tie\_breaker": 0.3,

"minimum\_should\_match": "50%"

}

}

}

**type 的值可以是：best\_fields，most\_fields以及cross\_fields**

上面是下面的语法缩写，使用 multi\_match实现：

GET /forum/article/\_search

{

"query": {

"dis\_max": {

"queries": [

{

"match": {

"title": {

"query": "java beginner",

"minimum\_should\_match": "50%",

"boost": 2

}

}

},

{

"match": {

"body": {

"query": "java beginner",

"minimum\_should\_match": "30%"

}

}

}

],

"tie\_breaker": 0.3

}

}

}

疑问：minimum\_should\_match，主要是用来干嘛的？

答案：去长尾，long tail

解释：长尾，比如你搜索5个关键词，但是很多结果是只匹配1个关键词的，其实跟你想要的结果相差甚远，这些结果就是长尾

minimum\_should\_match，控制搜索结果的精准度，只有匹配一定数量的关键词的数据，才能返回

### 案例-multi\_match-most fields

上面的小结中：从best-fields换成most-fields策略

best-fields策略：主要是说将某一个field匹配尽可能多的关键词的doc优先返回回来

most-fields策略：主要是说尽可能返回更多field匹配到某个关键词的doc，优先返回回来

接下来我们在 article 里面添加 subtitle 字段。首先建立 mapping，然后创建几条数据看看。

\*\* 建立 mapping

POST /forum/\_mapping/article

{

"properties": {

"sub\_title": {

"type": "string",

"analyzer": "english",

"fields": {

"std": {

"type": "string",

"analyzer": "standard"

}

}

}

}

}

\*\* 创建数据

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"sub\_title" : "learning more courses"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"sub\_title" : "learned a lot of course"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"sub\_title" : "we have a lot of fun"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"sub\_title" : "both of them are good"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"sub\_title" : "haha, hello world"} }

\*\* 查询

GET /forum/article/\_search

{

"query": {

"match": {

"sub\_title": "learning courses"

}

}

}

结果：

|  |
| --- |
| {  "took": 3,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 1.219939,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 1.219939,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 0.5063205,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ],  "tag\_cnt": 2,  "view\_cnt": 30,  "title": "this is java and elasticsearch blog",  "content": "i like to write best elasticsearch article",  "sub\_title": "learning more courses"  }  }  ]  }  } |

结果现象：sub\_title用的是enligsh analyzer，所以还原了单词

原因分析：因为如果我们用的是类似于english analyzer这种分词器的话，就会将单词还原为其最基本的形态，stemmer

learning --> learn

learned --> learn

courses --> course

sub\_titile: learning coureses --> learn course

{ "doc" : {"sub\_title" : "learned a lot of course"} }，就排在了{ "doc" : {"sub\_title" : "learning more courses"} }的前面

GET /forum/article/\_search

{

"query": {

"match": {

"sub\_title": "learning courses"

}

}

}

很绕。。。。我自己都觉得很绕

很多东西，你看文字就觉得很绕，然后用语言去表述，也很绕，但是我觉得，用语言去说，相对来说会好一点点

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "learning courses",

"type": "most\_fields",

"fields": [ "sub\_title", "sub\_title.std" ]

}

}

}

结果：

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 1.219939,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 1.219939,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 1.012641, 这个分数大幅度提升，但是还是没有上面那个高  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ],  "tag\_cnt": 2,  "view\_cnt": 30,  "title": "this is java and elasticsearch blog",  "content": "i like to write best elasticsearch article",  "sub\_title": "learning more courses"  }  }  ]  }  } |

你问我，具体的分数怎么算出来的，很难说，因为这个东西很复杂， 还不只是TF/IDF算法。因为不同的query，不同的语法，都有不同的计算score的细节。

**与best\_fields的区别**

（1）best\_fields，是对多个field进行搜索，挑选某个field匹配度最高的那个分数，同时在多个query最高分相同的情况下，在一定程度上考虑其他query的分数。简单来说，你对多个field进行搜索，就想搜索到某一个field尽可能包含更多关键字的数据

优点：通过best\_fields策略，以及综合考虑其他field，还有minimum\_should\_match支持，可以尽可能精准地将匹配的结果推送到最前面

缺点：除了那些精准匹配的结果，其他差不多大的结果，排序结果不是太均匀，没有什么区分度了

实际的例子：百度之类的搜索引擎，最匹配的到最前面，但是其他的就没什么区分度了

（2）most\_fields，综合多个field一起进行搜索，尽可能多地让所有field的query参与到总分数的计算中来，此时就会是个大杂烩，出现类似best\_fields案例最开始的那个结果，结果不一定精准，某一个document的一个field包含更多的关键字，但是因为其他document有更多field匹配到了，所以排在了前面；所以需要建立类似sub\_title.std这样的field，尽可能让某一个field精准匹配query string，贡献更高的分数，将更精准匹配的数据排到前面

优点：将尽可能匹配更多field的结果推送到最前面，整个排序结果是比较均匀的

缺点：可能那些精准匹配的结果，无法推送到最前面

实际的例子：wiki，明显的most\_fields策略，搜索结果比较均匀，但是的确要翻好几页才能找到最匹配的结果

### 案例-multi\_match-most\_fields策略进行cross-fileds search弊端大揭秘

cross-fields搜索，一个唯一标识，跨了多个field。比如一个人，标识，是姓名；一个建筑，它的标识是地址。姓名可以散落在多个field中，比如first\_name和last\_name中，地址可以散落在country，province，city中。

跨多个field搜索一个标识，比如搜索一个人名，或者一个地址，就是cross-fields搜索

初步来说，如果要实现，可能用most\_fields比较合适。因为best\_fields是优先搜索单个field最匹配的结果，cross-fields本身就不是一个field的问题了。

我们先添加 author\_first\_name的一些数据

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"author\_first\_name" : "Peter", "author\_last\_name" : "Smith"} }

{ "update": { "\_id": "2"} }

{ "doc" : {"author\_first\_name" : "Smith", "author\_last\_name" : "Williams"} }

{ "update": { "\_id": "3"} }

{ "doc" : {"author\_first\_name" : "Jack", "author\_last\_name" : "Ma"} }

{ "update": { "\_id": "4"} }

{ "doc" : {"author\_first\_name" : "Robbin", "author\_last\_name" : "Li"} }

{ "update": { "\_id": "5"} }

{ "doc" : {"author\_first\_name" : "Tonny", "author\_last\_name" : "Peter Smith"} }

先使用 most\_fields搜索，结果不是很理想

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "most\_fields",

"fields": [ "author\_first\_name", "author\_last\_name" ]

}

}

}

疑问：应该是Peter Smith(ID=5)，是完美匹配author\_first\_name，但是匹配到了Smith(ID=1)，这时候它的分数很高，为什么啊？？？

解释：因为IDF分数高，IDF分数要高，那么这个匹配到的term（Smith），在所有doc中的出现频率要低，author\_first\_name field中，Smith就出现过1次

Peter Smith这个人，doc 1，Smith在author\_last\_name中，但是author\_last\_name出现了两次Smith，所以导致doc 1的IDF分数较低

不要有过多的疑问，一定是上面这样吗？哈哈哈，分数计算很细的，除非是源码计算逻辑贡献者。这里说的知识一个大概的规律。

结果：

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 0.6931472,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 0.6931472,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course",  "author\_first\_name": "Smith",  "author\_last\_name": "Williams"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 0.5753642,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ],  "tag\_cnt": 2,  "view\_cnt": 30,  "title": "this is java and elasticsearch blog",  "content": "i like to write best elasticsearch article",  "sub\_title": "learning more courses",  "author\_first\_name": "Peter",  "author\_last\_name": "Smith"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "5",  "\_score": 0.51623213,  "\_source": {  "articleID": "DHJK-B-1395-#Ky5",  "userID": 3,  "hidden": false,  "postDate": "2017-03-01",  "tag": [  "elasticsearch"  ],  "tag\_cnt": 1,  "view\_cnt": 10,  "title": "this is spark blog",  "content": "spark is best big data solution based on scala ,an programming language similar to java",  "sub\_title": "haha, hello world",  "author\_first\_name": "Tonny",  "author\_last\_name": "Peter Smith"  }  }  ]  }  } |

问题1：只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc

问题2：most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果

问题3：TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面

### 使用copy\_to定制组合 field 解决 cross-fields搜索弊端

上一讲，我们其实说了，用most\_fields策略，去实现cross-fields搜索，有3大弊端，而且搜索结果也显示出了这3大弊端

**第一个办法：**用copy\_to，将多个field组合成一个field

问题其实就出在有多个field，有多个field以后，就很尴尬，我们只要想办法将一个标识跨在多个field的情况，合并成一个field即可。比如说，一个人名，本来是first\_name，last\_name，现在合并成一个full\_name，不就ok了吗。。。。。

PUT /forum/\_mapping/article

{

"properties": {

"new\_author\_first\_name": {

"type": "string",

"copy\_to": "new\_author\_full\_name"

},

"new\_author\_last\_name": {

"type": "string",

"copy\_to": "new\_author\_full\_name"

},

"new\_author\_full\_name": {

"type": "string"

}

}

}

用了这个copy\_to语法之后，就可以将多个字段的值拷贝到一个字段中，并建立倒排索引

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"new\_author\_first\_name" : "Peter", "new\_author\_last\_name" : "Smith"} } --> Peter Smith

{ "update": { "\_id": "2"} }

{ "doc" : {"new\_author\_first\_name" : "Smith", "new\_author\_last\_name" : "Williams"} } --> Smith Williams

{ "update": { "\_id": "3"} }

{ "doc" : {"new\_author\_first\_name" : "Jack", "new\_author\_last\_name" : "Ma"} } --> Jack Ma

{ "update": { "\_id": "4"} }

{ "doc" : {"new\_author\_first\_name" : "Robbin", "new\_author\_last\_name" : "Li"} } --> Robbin Li

{ "update": { "\_id": "5"} }

{ "doc" : {"new\_author\_first\_name" : "Tonny", "new\_author\_last\_name" : "Peter Smith"} } --> Tonny Peter Smith

接着查询：

GET /forum/article/\_search

{

"query": {

"match": {

"new\_author\_full\_name": "Peter Smith"

}

}

}

很无奈，很多时候，我们很难复现。比如官网也会给一些例子，说用什么什么文本，怎么怎么搜索，是怎么怎么样的效果。es版本在不断迭代，这个打分的算法也在不断的迭代。所以我们其实很难说，对类似这几讲讲解的best\_fields，most\_fields，cross\_fields，完全复现出来应有的场景和效果。

更多的把原理和知识点给大家讲解清楚，带着大家演练一遍怎么操作的，做一下实验

期望的是说，比如大家自己在开发搜索应用的时候，碰到需要best\_fields的场景，知道怎么做，知道best\_fields的原理，可以达到什么效果；碰到most\_fields的场景，知道怎么做，以及原理；碰到搜搜cross\_fields标识的场景，知道怎么做，知道原理是什么，效果是什么。。。。

问题1：只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc --> 解决，最匹配的document被最先返回

问题2：most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果 --> 解决，可以使用minimum\_should\_match去掉长尾数据

问题3：TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面 --> 解决，Smith和Peter在一个field了，所以在所有document中出现的次数是均匀的，不会有极端的偏差

### 使用原生 cross-fields技术解决搜索弊端

GET /forum/article/\_search

{

"query": {

"multi\_match": {

"query": "Peter Smith",

"type": "cross\_fields",

"operator": "and",

"fields": ["author\_first\_name", "author\_last\_name"]

}

}

}

**问题1：**只是找到尽可能多的field匹配的doc，而不是某个field完全匹配的doc

解决：要求每个term都必须在任何一个field中出现

如搜索Peter，Smith

要求Peter必须在author\_first\_name或author\_last\_name中出现

要求Smith必须在author\_first\_name或author\_last\_name中出现

Peter Smith可能是横跨在多个field中的，所以必须要求每个term都在某个field中出现，组合起来才能组成我们想要的标识，完整的人名

原来most\_fiels，可能像Smith Williams也可能会出现，因为most\_fields要求只是任何一个field匹配了就可以，匹配的field越多，分数越高

**问题2：**most\_fields，没办法用minimum\_should\_match去掉长尾数据，就是匹配的特别少的结果

解决：既然每个term都要求出现，长尾肯定被去除掉了

java hadoop spark --> 这3个term都必须在任何一个field出现了

比如有的document，只有一个field中包含一个java，那就被干掉了，作为长尾就没了

**问题3：**TF/IDF算法，比如Peter Smith和Smith Williams，搜索Peter Smith的时候，由于first\_name中很少有Smith的，所以query在所有document中的频率很低，得到的分数很高，可能Smith Williams反而会排在Peter Smith前面  
解决：计算IDF的时候，将每个query在每个field中的IDF都取出来，取最小值，就不会出现极端情况下的极大值了

Peter Smith

Peter

Smith

Smith，在author\_first\_name这个field中，在所有doc的这个Field中，出现的频率很低，导致IDF分数很高；Smith在所有doc的author\_last\_name field中的频率算出一个IDF分数，因为一般来说last\_name中的Smith频率都较高，所以IDF分数是正常的，不会太高；然后对于Smith来说，会取两个IDF分数中，较小的那个分数。就不会出现IDF分过高的情况。

### 案例-phase matching搜索技术掌握

1、什么是近似匹配

举个例子，这里有两个句子：

java is my favourite programming language, and I also think spark is a very good big data system.

java spark are very related, because scala is spark's programming language and scala is also based on jvm like java.

我们使用match query，搜索"java spark"

{

"match": {

"content": "java spark"

}

}

match query，只能搜索到包含java和spark的document，但是不知道java和spark是不是离的很近

包含java或包含spark，或包含java和spark的doc，都会被返回回来。我们其实并不知道哪个doc，java和spark距离的比较近。如果我们就是希望搜索java spark，中间不能插入任何其他的字符，那这个时候match去做全文检索，能搞定我们的需求吗？答案是，搞不定。

如果我们要尽量让java和spark离的很近的document优先返回，要给它一个更高的relevance score，这就涉及到了proximity match，近似匹配

如果说，要实现两个需求：

1、java spark，就靠在一起，中间不能插入任何其他字符，就要搜索出来这种doc

2、java spark，但是要求，java和spark两个单词靠的越近，doc的分数越高，排名越靠前

要实现上述两个需求，用match做全文检索，是搞不定的，必须得用proximity match，近似匹配，命令如下：

**phrase match：短语匹配**

**proximity match：近似匹配**

这一讲，要学习的是phrase match，就是仅仅搜索出java和spark靠在一起的那些doc，比如有个doc，是java use'd spark，不行。必须是比如java spark are very good friends，是可以搜索出来的。

phrase match，就是要去将多个term作为一个短语，一起去搜索，只有包含这个短语的doc才会作为结果返回。不像是match，java spark，java的doc也会返回，spark的doc也会返回。

2、match\_phrase

先使用 match 搜索测试一下，发现是包含 java 或者 spark 的都会返回

GET /forum/article/\_search

{

"query": {

"match": {

"content": "java spark"

}

}

}

单单包含java的doc也返回了，不是我们想要的结果

将一个doc的content设置为恰巧包含java spark这个短语

POST /forum/article/5/\_update

{

"doc": {

"content": "spark is best big data solution based on scala ,an programming language similar to java spark"

}

}

match\_phrase语法：

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": "java spark"

}

}

}

成功了，只有包含java spark这个短语的doc才返回了，只包含java的doc不会返回

3、term position

了解 term position 的概念。如下的例子：

hello world, java spark doc1

hi, spark java doc2

经过分词器以后会变成下面的样子：

hello doc1(0)

wolrd doc1(1)

java doc1(2) doc2(2)

spark doc1(3) doc2(1)

了解什么是分词后的position

GET \_analyze

{

"text": "hello world, java spark",

"analyzer": "standard"

}

4、match\_phrase的基本原理

match\_phrase的原理就是根据索引中的position来判断。

hello world, java spark doc1

hi, spark java doc2

hello doc1(0)

wolrd doc1(1)

java doc1(2) doc2(2)

spark doc1(3) doc2(1)

java spark --> match phrase

java spark --> java和spark

java --> doc1(2) doc2(2)

spark --> doc1(3) doc2(1)

要找到每个term都在的一个共有的那些doc，就是要求一个doc，必须包含每个term，才能拿出来继续计算

doc1 --> java和spark --> spark position恰巧比java大1 --> java的position是2，spark的position是3，恰好满足条件

doc1符合条件

doc2 --> java和spark --> java position是2，spark position是1，spark position比java position小1，而不是大1 --> 光是position就不满足，那么doc2不匹配

必须理解这块原理！！！！

因为后面的proximity match就是原理跟这个一模一样！！！

### 实验-slop 参数实现近似匹配以及原理剖析和相关实验

语法格式：

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "java spark",

"slop": 1

}

}

}

}

slop的含义是什么？

query string，搜索文本，中的几个term，要经过几次移动才能与一个document匹配，这个移动的次数，就是slop

实际举例，一个query string经过几次移动之后可以匹配到一个document，然后设置slop

doc：hello world, java is very good, spark is also very good.

对于搜索java spark，使用match phrase是搜不到的，下面来说一下 slop 的例子：

如果我们指定了slop，那么就允许java spark进行移动，来尝试与doc进行匹配

java is very good spark is

java spark

java --> spark

java --> spark

java --> spark

这里的slop，就是3，因为java spark这个短语，spark移动了3次，就可以跟一个doc匹配上了

slop的含义，不仅仅是说一个query string terms移动几次，跟一个doc匹配上。一个query string terms，最多可以移动几次去尝试跟一个doc匹配上

slop，设置的是3，那么就ok

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"title": {

"query": "java spark",

"slop": 3

}

}

}

}

就可以把刚才那个doc匹配上，那个doc会作为结果返回

但是如果slop设置的是2，那么java spark，spark最多只能移动2次，此时跟doc是匹配不上的，那个doc是不会作为结果返回的

做实验，验证slop的含义，这里进行搜索 "spark data"

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "spark data",

"slop": 3

}

}

}

}

搜索出来的doc是：spark is best big data solution based on scala ,an programming language similar to java spark

现象描述：

spark data

--> data

--> data

spark --> data

接下来如果换成"data spark"的话，3，4 都不行。5 就可以了。为啥？

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "data spark",

"slop": 5

}

}

}

}

现象解释：

spark is best big data

data spark

--> data/spark

spark <--data

spark --> data

spark --> data

spark --> data

slop搜索下，关键词离的越近，relevance score就会越高，做实验说明。。。

直接搜索出来的结果：

|  |
| --- |
| {  "took": 4,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1.3728157,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 1.3728157,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course",  "author\_first\_name": "Smith",  "author\_last\_name": "Williams",  "new\_author\_last\_name": "Williams",  "new\_author\_first\_name": "Smith"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "5",  "\_score": 0.5753642,  "\_source": {  "articleID": "DHJK-B-1395-#Ky5",  "userID": 3,  "hidden": false,  "postDate": "2017-03-01",  "tag": [  "elasticsearch"  ],  "tag\_cnt": 1,  "view\_cnt": 10,  "title": "this is spark blog",  "content": "spark is best big data solution based on scala ,an programming language similar to java spark",  "sub\_title": "haha, hello world",  "author\_first\_name": "Tonny",  "author\_last\_name": "Peter Smith",  "new\_author\_last\_name": "Peter Smith",  "new\_author\_first\_name": "Tonny"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "1",  "\_score": 0.28582606,  "\_source": {  "articleID": "XHDK-A-1293-#fJ3",  "userID": 1,  "hidden": false,  "postDate": "2017-01-01",  "tag": [  "java",  "hadoop"  ],  "tag\_cnt": 2,  "view\_cnt": 30,  "title": "this is java and elasticsearch blog",  "content": "i like to write best elasticsearch article",  "sub\_title": "learning more courses",  "author\_first\_name": "Peter",  "author\_last\_name": "Smith",  "new\_author\_last\_name": "Smith",  "new\_author\_first\_name": "Peter"  }  }  ]  }  } |

从上面的数据可以看出 java best 都有，我们就搜"java best"

GET /forum/article/\_search

{

"query": {

"match\_phrase": {

"content": {

"query": "java best",

"slop": 15

}

}

}

}

结果：

|  |
| --- |
| {  "took": 3,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0.65380025,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 0.65380025,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course",  "author\_first\_name": "Smith",  "author\_last\_name": "Williams",  "new\_author\_last\_name": "Williams",  "new\_author\_first\_name": "Smith"  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "5",  "\_score": 0.07111243,  "\_source": {  "articleID": "DHJK-B-1395-#Ky5",  "userID": 3,  "hidden": false,  "postDate": "2017-03-01",  "tag": [  "elasticsearch"  ],  "tag\_cnt": 1,  "view\_cnt": 10,  "title": "this is spark blog",  "content": "spark is best big data solution based on scala ,an programming language similar to java spark",  "sub\_title": "haha, hello world",  "author\_first\_name": "Tonny",  "author\_last\_name": "Peter Smith",  "new\_author\_last\_name": "Peter Smith",  "new\_author\_first\_name": "Tonny"  }  }  ]  }  } |

其实，加了slop的phrase match，就是proximity match，近似匹配

1、java spark，短语，doc，phrase match

2、java spark，可以有一定的距离，但是靠的越近，越先搜索出来，proximity match

### 实战-混合使用 match 和近似匹配实现召回率与精准度的平衡

什么是召回率？

答案：比如你搜索一个java spark，总共有100个doc，能返回多少个doc作为结果，就是召回率，recall

什么是精准度？

答案：比如你搜索一个java spark，能不能尽可能让包含java spark，或者是java和spark离的很近的doc，排在最前面，precision

直接用match\_phrase短语搜索，会导致必须所有term都在doc field中出现，而且距离在slop限定范围内，才能匹配上

match phrase，proximity match，要求doc必须包含所有的term，才能作为结果返回；如果某一个doc可能就是有某个term没有包含，那么就无法作为结果返回

java spark --> hello world java --> 就不能返回了

java spark --> hello world, java spark --> 才可以返回

近似匹配的时候，召回率比较低，精准度太高了

但是有时可能我们希望的是匹配到几个term中的部分，就可以作为结果出来，这样可以提高召回率。同时我们也希望用上match\_phrase根据距离提升分数的功能，让几个term距离越近分数就越高，优先返回

就是优先满足召回率，意思，java spark，包含java的也返回，包含spark的也返回，包含java和spark的也返回；同时兼顾精准度，就是包含java和spark，同时java和spark离的越近的doc排在最前面

此时可以用bool组合match query和match\_phrase query一起，来实现上述效果

GET /forum/article/\_search

{

"query": {

"bool": {

"must": {

"match": {

"title": {

"query": "java spark" --> java或spark或java spark，java和spark靠前，但是没法区分java和spark的距离，也许java和spark靠的很近，但是没法排在最前面

}

}

},

"should": {

"match\_phrase": { --> 在slop以内，如果java spark能匹配上一个doc，那么就会对doc贡献自己的relevance score，如果java和spark靠的越近，那么就分数越高

"title": {

"query": "java spark",

"slop": 50

}

}

}

}

}

}

首先看直接匹配 java 和 spark的

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java spark"

}

}

]

}

}

}

结果：

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0.68640786,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 0.68640786,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course",  "author\_first\_name": "Smith",  "author\_last\_name": "Williams",  "new\_author\_last\_name": "Williams",  "new\_author\_first\_name": "Smith",  "followers": [  "Tom",  "Jack"  ]  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "5",  "\_score": 0.68324494,  "\_source": {  "articleID": "DHJK-B-1395-#Ky5",  "userID": 3,  "hidden": false,  "postDate": "2017-03-01",  "tag": [  "elasticsearch"  ],  "tag\_cnt": 1,  "view\_cnt": 10,  "title": "this is spark blog",  "content": "spark is best big data solution based on scala ,an programming language similar to java spark",  "sub\_title": "haha, hello world",  "author\_first\_name": "Tonny",  "author\_last\_name": "Peter Smith",  "new\_author\_last\_name": "Peter Smith",  "new\_author\_first\_name": "Tonny",  "followers": [  "Jack",  "Robbin Li"  ]  }  }  ]  }  } |

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java spark"

}

}

],

"should": [

{

"match\_phrase": {

"content": {

"query": "java spark",

"slop": 50

}

}

}

]

}

}

}

结果：

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 1.258609,  "hits": [  {  "\_index": "forum",  "\_type": "article",  "\_id": "5",  "\_score": 1.258609,  "\_source": {  "articleID": "DHJK-B-1395-#Ky5",  "userID": 3,  "hidden": false,  "postDate": "2017-03-01",  "tag": [  "elasticsearch"  ],  "tag\_cnt": 1,  "view\_cnt": 10,  "title": "this is spark blog",  "content": "spark is best big data solution based on scala ,an programming language similar to java spark",  "sub\_title": "haha, hello world",  "author\_first\_name": "Tonny",  "author\_last\_name": "Peter Smith",  "new\_author\_last\_name": "Peter Smith",  "new\_author\_first\_name": "Tonny",  "followers": [  "Jack",  "Robbin Li"  ]  }  },  {  "\_index": "forum",  "\_type": "article",  "\_id": "2",  "\_score": 0.68640786,  "\_source": {  "articleID": "KDKE-B-9947-#kL5",  "userID": 1,  "hidden": false,  "postDate": "2017-01-02",  "tag": [  "java"  ],  "tag\_cnt": 1,  "view\_cnt": 50,  "title": "this is java blog",  "content": "i think java is the best programming language",  "sub\_title": "learned a lot of course",  "author\_first\_name": "Smith",  "author\_last\_name": "Williams",  "new\_author\_last\_name": "Williams",  "new\_author\_first\_name": "Smith",  "followers": [  "Tom",  "Jack"  ]  }  }  ]  }  } |

### 原理-rescoring 机制优化近似匹配搜索的性能

match和phrase match(proximity match)区别

match --> 只要简单的匹配到了一个term，就可以理解将term对应的doc作为结果返回，扫描倒排索引，扫描到了就ok

phrase match --> 首先扫描到所有term的doc list; 找到包含所有term的doc list; 然后对每个doc都计算每个term的position，是否符合指定的范围; slop，需要进行复杂的运算，来判断能否通过slop移动，匹配一个doc

match query的性能比phrase match和proximity match（有slop）要高很多。因为后两者都要计算position的距离。

match query比phrase match的性能要高10倍，比proximity match的性能要高20倍。

但是别太担心，因为es的性能一般都在毫秒级别，match query一般就在几毫秒，或者几十毫秒，而phrase match和proximity match的性能在几十毫秒到几百毫秒之间，所以也是可以接受的。

优化proximity match的性能，一般就是减少要进行proximity match搜索的document数量。主要思路就是，用match query先过滤出需要的数据，然后再用proximity match来根据term距离提高doc的分数，同时proximity match只针对每个shard的分数排名前n个doc起作用，来重新调整它们的分数，这个过程称之为rescoring，重计分。因为一般用户会分页查询，只会看到前几页的数据，所以不需要对所有结果进行proximity match操作。

用我们刚才的说法，match + proximity match同时实现召回率和精准度

默认情况下，match也许匹配了1000个doc，proximity match全都需要对每个doc进行一遍运算，判断能否slop移动匹配上，然后去贡献自己的分数

但是很多情况下，match出来也许1000个doc，其实用户大部分情况下是分页查询的，所以可能最多只会看前几页，比如一页是10条，最多也许就看5页，就是50条

proximity match只要对前50个doc进行slop移动去匹配，去贡献自己的分数即可，不需要对全部1000个doc都去进行计算和贡献分数

上面说的这个过程就是：rescore，叫重打分的一个过程。

match：1000个doc，其实这时候每个doc都有一个分数了; proximity match，前50个doc，进行rescore，重打分，即可; 让前50个doc，term举例越近的，排在越前面

GET /forum/article/\_search

{

"query": {

"match": {

"content": "java spark"

}

},

"rescore": {

"window\_size": 50,

"query": {

"rescore\_query": {

"match\_phrase": {

"content": {

"query": "java spark",

"slop": 50

}

}

}

}

}

}

### 实战-前缀搜索、通配符搜索、正则搜索等技术

使用关键词：**prefix、wildcard、regexp**

1、前缀搜索

C3D0-KD345

C3K5-DFG65

C4I8-UI365

C3 --> 上面这两个都搜索出来 --> 根据字符串的前缀去搜索

不用帖子的案例背景，因为比较简单，直接用自己手动建的新索引，给大家演示一下就可以了

创建 index 的时候定义mapping：

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"title": {

"type": "keyword"

}

}

}

}

}

前缀搜索：

GET my\_index/my\_type/\_search

{

"query": {

"prefix": {

"title": {

"value": "C3"

}

}

}

}

2、前缀搜索的原理

prefix query不计算relevance score，与prefix filter唯一的区别就是，filter会cache bitset

扫描整个倒排索引，举例说明

前缀越短，要处理的doc越多，性能越差，尽可能用长前缀搜索

前缀搜索，它是怎么执行的？性能为什么差呢？

我这里如果是match的话，对下面的 doc 进行距离：

C3-D0-KD345

C3-K5-DFG65

C4-I8-UI365

如果是进行全文检索，那么每个字符串都需要被分词，如下的样子：

c3 doc1,doc2

d0

kd345

k5

dfg65

c4

i8

ui365

c3 --> 扫描倒排索引 --> 一旦扫描到c3，就可以停了，因为带c3的就2个doc，已经找到了 --> 没有必要继续去搜索其他的term了

所以，结论是：match性能往往是很高的

如果是下面的 doc不分词 的话：

C3-D0-KD345

C3-K5-DFG65

C4-I8-UI365

c3 --> 先扫描到了C3-D0-KD345，很棒，找到了一个前缀带c3的字符串 --> 还是要继续搜索的，因为后面还有一个C3-K5-DFG65，也许还有其他很多的前缀带c3的字符串 --> 你扫描到了一个前缀匹配的term，不能停，必须继续搜索 --> 直到扫描完整个的倒排索引，才能结束

因为实际场景中，可能有些场景是全文检索解决不了的

C3D0-KD345

C3K5-DFG65

C4I8-UI365

c3d0

kd345

c3 --> match --> 扫描整个倒排索引，能找到吗？找不到，所以这个时候用 prefix 搜索。

c3 --> 只能用prefix

但是prefix性能很差

3、通配符搜索

跟前缀搜索类似，功能更加强大

C3D0-KD345

C3K5-DFG65

C4I8-UI365

5字符-D任意个字符5

5?-\*5：通配符去表达更加复杂的模糊搜索的语义

GET my\_index/my\_type/\_search

{

"query": {

"wildcard": {

"title": {

"value": "C?K\*5"

}

}

}

}

?：任意字符

\*：0个或任意多个字符

性能一样差，必须扫描整个倒排索引，才ok

4、正则搜索

GET /my\_index/my\_type/\_search

{

"query": {

"regexp": {

"title": "C[0-9].+"

}

}

}

C[0-9].+

[0-9]：指定范围内的数字

[a-z]：指定范围内的字母

.：一个字符

+：前面的正则表达式可以出现一次或多次

wildcard和regexp，与prefix原理一致，都会扫描整个索引，性能很差

主要是给大家介绍一些高级的搜索语法。在实际应用中，能不用尽量别用。性能太差了。

### 实战-搜索推荐-match\_phase\_prefix实现(search-time)

搜索推荐，search as you type，搜索提示，解释一下什么意思

如进行hello w --> 搜索，就会显示如下的结果：

hello world

hello we

hello win

hello wind

这就是搜索推荐的功能

如在百度上去去搜索： elas ，就会显示： elasticsearch --> elasticsearch权威指南

GET /my\_index/my\_type/\_search

{

"query": {

"match\_phrase\_prefix": {

"title": "hello d"

}

}

}

原理跟match\_phrase类似，唯一的区别，就是把最后一个term作为前缀去搜索

hello就是去进行match，搜索对应的doc

w，会作为前缀，去扫描整个倒排索引，找到所有w开头的doc

然后找到所有doc中，即包含hello，又包含w开头的字符的doc

根据你的slop去计算，看在slop范围内，能不能让hello w，正好跟doc中的hello和w开头的单词的position相匹配

也可以指定slop，但是只有最后一个term会作为前缀

max\_expansions：指定prefix最多匹配多少个term，超过这个数量就不继续匹配了，限定性能

默认情况下，前缀要扫描所有的倒排索引中的term，去查找w打头的单词，但是这样性能太差。可以用max\_expansions限定，w前缀最多匹配多少个term，就不再继续搜索倒排索引了。

尽量不要用，因为，最后一个前缀始终要去扫描大量的索引，性能可能会很差

### 实战-搜索推荐-ngram 分词机制实现(index-time)

1、ngram和index-time搜索推荐原理

什么是ngram？

解答：对于单词：quick，5种长度下的ngram如下所示：

ngram length=1，q u i c k

ngram length=2，qu ui ic ck

ngram length=3，qui uic ick

ngram length=4，quic uick

ngram length=5，quick

那么什么是**edge ngram**？

同样对于单词：quick，anchor首字母后进行ngram，会有如下情况：

q

qu

qui

quic

quick

**使用edge ngram将每个单词都进行进一步的分词切分，用切分后的ngram来实现前缀搜索推荐功能**

下面进行解释：

如对"hello world"的单词，会进行如下的切分：

h

he

hel

hell

hello doc1,doc2

w doc1,doc2

wo

wor

worl

world

e doc2

helloworld

min ngram = 1

max ngram = 3

h

he

hel

hello --> hello，doc1

w --> w，doc1

我们搜索hello w 的时候，就包含 hello 和 w，而且position也匹配，所以，ok，doc1返回，hello world

搜索的时候，不用再根据一个前缀，然后扫描整个倒排索引了; 简单的拿前缀去倒排索引中匹配即可，如果匹配上了，那么就好了; match，全文检索

2、实验一下ngram

创建索引的需要设置 filter 为 edge\_ngram类型的 autocomplete\_filter

PUT /my\_index

{

"settings": {

"analysis": {

"filter": {

"autocomplete\_filter": {

"type": **"edge\_ngram"**,

"min\_gram": 1,

"max\_gram": 20

}

},

"analyzer": {

"autocomplete": {

"type": "custom",

"tokenizer": "standard",

"filter": [

"lowercase",

"autocomplete\_filter"

]

}

}

}

}

}

分词测试

GET /my\_index/\_analyze

{

"analyzer": "autocomplete",

"text": "quick brown"

}

PUT /my\_index/\_mapping/my\_type

{

"properties": {

"title": {

"type": "string",

"analyzer": "autocomplete",

"search\_analyzer": "standard"

}

}

}

hello world

h

he

hel

hell

hello

w

wo

wor

worl

world

hello w

h

he

hel

hell

hello

w

hello w --> hello --> w

GET /my\_index/my\_type/\_search

{

"query": {

"match\_phrase": {

"title": "hello w"

}

}

}

如果用match，只有hello的也会出来，全文检索，只是分数比较低

推荐使用match\_phrase，要求每个term都有，而且position刚好靠着1位，符合我们的期望的

### 解密-TF-IDF算法以及向量空间模型算法

1、boolean model

类似and这种逻辑操作符，先过滤出包含指定term的doc

比如说查询"hello world" --> ES 就过滤 --> hello / world / hello & world

包括使用bool查询的 --> must/must not/should --> 过滤 --> 包含 / 不包含 / 可能包含

过滤出的doc --> 不打分数 --> 正或反 true or false --> 为了减少后续要计算的doc的数量，提升性能

2、TF/IDF

单个term在doc中的分数

如搜索 hello world。（针对的是doc.content）

doc1: java is my favourite programming language, hello world !!!

doc2: hello java, you are very good, oh hello world!!!

首先是hello对doc1的评分

\*\* TF: term frequency

找到hello在doc1中出现了几次，1次，会根据出现的次数给个分数

一个term在一个doc中，出现的次数越多，那么最后给的相关度评分就会越高

\*\* IDF：inversed document frequency

找到hello在所有的doc中出现的次数，3次

一个term在所有的doc中，出现的次数越多，那么最后给的相关度评分就会越低

\*\* length norm

hello搜索的那个field的长度，field长度越长，给的相关度评分越低; field长度越短，给的相关度评分越高

最后，会将hello这个term，对doc1的分数，综合TF，IDF，length norm，计算出来一个综合性的分数

hello world --> doc1 --> hello对doc1的分数，world对doc1的分数 --> 但是最后hello world query要对doc1有一个总的分数 --> vector space model

3、vector space model

意思就是：多个term对一个doc的总分数

hello world --> es会根据hello world在所有doc中的评分情况，计算出一个query vector，query向量

hello这个term，给的基于所有doc的一个评分就是2

world这个term，给的基于所有doc的一个评分就是5

这里有一个query vector：[2, 5]

doc vector，3个doc，一个包含1个term，一个包含另一个term，一个包含2个term

假设现在有3个doc：

doc1：包含hello --> [2, 0]

doc2：包含world --> [0, 5]

doc3：包含hello, world --> [2, 5]

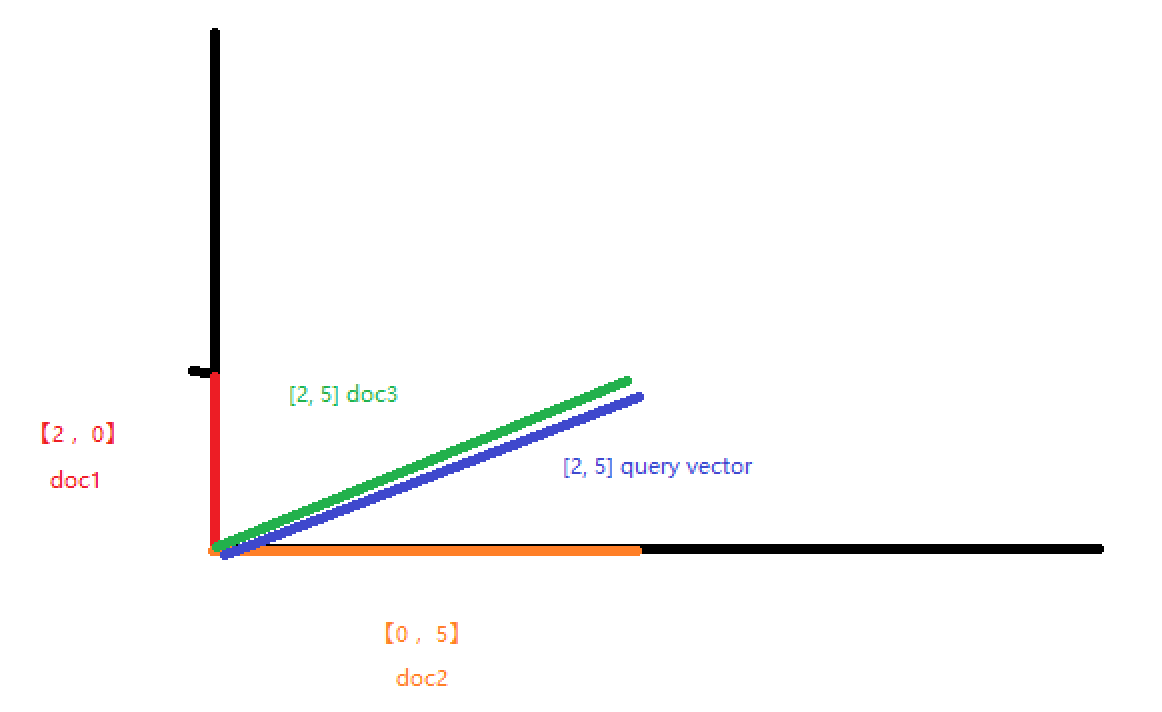
会给每一个doc，拿每个term计算出一个分数来，hello有一个分数，world有一个分数，再拿所有term的分数组成一个doc vector

画在一个图中，取每个doc vector对query vector的弧度，给出每个doc对多个term的总分数

每个doc vector计算出对query vector的弧度，最后基于这个弧度给出一个doc相对于query中多个term的总分数

弧度越大，分数月底; 弧度越小，分数越高

如果是多个term，那么就是线性代数来计算，无法用图表示



### 解密-Lucene 的相关度分数算法

上一讲主要是：boolean model、TF/IDF、vector space model

参考链接：

2.x 版本：

https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html

这一讲我们深入讲解TF/IDF算法，在lucene中，底层，到底进行TF/IDF算法计算的一个完整的公式是什么？

0、boolean model

如我们去搜索: hello world

"match": {

"title": "hello world"

}

他会转换成如下这种模式：

"bool": {

"should": [

{

"match": {

"title": "hello"

}

},

{

"natch": {

"title": "world"

}

}

]

}

普通multivalue搜索，转换为bool搜索，boolean model

1、lucene practical scoring function

**practical scoring function**，来计算一个query对一个doc的分数的公式，该函数会使用一个公式来计算

下面就是这个公式

score(q,d) =

queryNorm(q)

・ coord(q,d)

・ ∑ (

tf(t in d)

・ idf(t)2

・ t.getBoost()

・ norm(t,d)

) (t in q)

score(q,d) score(q,d) is the relevance score of document d for query q.

这个公式的最终结果：就是说是一个query（叫做q），对一个doc（叫做d）的最终的总评分

queryNorm(q) is the query normalization factor (new).

queryNorm，是用来让一个doc的分数处于一个合理的区间内，不要太离谱，举个例子，一个doc分数是10000，一个doc分数是0.1，你们说好不好，肯定不好

coord(q,d) is the coordination factor (new).

简单来说，就是对更加匹配的doc，进行一些分数上的成倍的奖励

The sum of the weights for each term t in the query q for document d.

∑：求和的符号

∑ (t in q)：query中每个term，query = hello world，query中的term就包含了hello和world

query中每个term对doc的分数，进行求和，多个term对一个doc的分数，组成一个vector space，然后计算吗，就在这一步

tf(t in d) is the term frequency for term t in document d.

计算每一个term对doc的分数的时候，就是TF/IDF算法

idf(t) is the inverse document frequency for term t.

t.getBoost() is the boost that has been applied to the query (new).

norm(t,d) is the field-length norm, combined with the index-time field-level boost, if any. (new).

2、query normalization factor

queryNorm = 1 / √sumOfSquaredWeights

sumOfSquaredWeights = 所有term的IDF分数之和，开一个平方根，然后做一个平方根分之1

主要是为了将分数进行规范化 --> 开平方根，首先数据就变小了 --> 然后还用1去除以这个平方根，分数就会很小 --> 1.几 / 零点几

分数就不会出现几万，几十万，那样的离谱的分数

3、query coodination

奖励那些匹配更多字符的doc更多的分数

Document 1 with hello → score: 1.5

Document 2 with hello world → score: 3.0

Document 3 with hello world java → score: 4.5

Document 1 with hello → score: 1.5 \* 1 / 3 = 0.5

Document 2 with hello world → score: 3.0 \* 2 / 3 = 2.0

Document 3 with hello world java → score: 4.5 \* 3 / 3 = 4.5

把计算出来的总分数 \* 匹配上的term数量 / 总的term数量，让匹配不同term/query数量的doc，分数之间拉开差距

4、field level boost

### 实战-四种常见的相关度分数优化方法

之前两节课，我觉得已经很了解整个es的相关度评分的算法了，算法思想，TF/IDF，vector model，boolean model; 实际的公式，query norm，query coordination，boost

本讲主题：对相关度评分进行调节和优化的常见的4种方法

1、query-time boost

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": {

"query": "java spark",

"boost": 2

}

}

},

{

"match": {

"content": "java spark"

}

}

]

}

}

}

2、重构查询结构

重构查询结果，在es新版本中，影响越来越小了。一般情况下，没什么必要的话，大家不用也行。

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"content": "java"

}

},

{

"match": {

"content": "spark"

}

},

{

"bool": {

"should": [

{

"match": {

"content": "solution"

}

},

{

"match": {

"content": "beginner"

}

}

]

}

}

]

}

}

}

3、negative boost

搜索包含java，不包含spark的doc，但是这样子很死板

搜索包含java，尽量不包含spark的doc，如果包含了spark，不会说排除掉这个doc，而是说将这个doc的分数降低

包含了negative term的doc，分数乘以negative boost，分数降低

如下使用 match，和 match not 就比较绝对，我不想这么觉得

GET /forum/article/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"content": "java"

}

}

],

"must\_not": [

{

"match": {

"content": "spark"

}

}

]

}

}

}

就可以使用 positive 和 negative。并且给 negative 设置一个 negative\_boost的分数。

GET /forum/article/\_search

{

"query": {

"boosting": {

**"positive":** {

"match": {

"content": "java"

}

},

**"negative":** {

"match": {

"content": "spark"

}

},

**"negative\_boost": 0.2**

}

}

}

negative的doc，会乘以negative\_boost，降低分数

4、constant\_score

如果你压根儿不需要相关度评分，直接走constant\_score加filter，所有的doc分数都是1，没有评分的概念了

GET /forum/article/\_search

{

"query": {

"bool": {

"should": [

{

"constant\_score": {

"query": {

"match": {

"title": "java"

}

}

}

},

{

"constant\_score": {

"query": {

"match": {

"title": "spark"

}

}

}

}

]

}

}

}

### 实战-function\_score自定义相关度分数算法

我们可以做到自定义一个function\_score函数，自己将某个field的值，跟es内置算出来的分数进行运算，然后由自己指定的field来进行分数的增强

这里做一个需求：给所有的帖子数据增加follower数量

首先添加 follower 数据：

POST /forum/article/\_bulk

{ "update": { "\_id": "1"} }

{ "doc" : {"follower\_num" : 5} }

{ "update": { "\_id": "2"} }

{ "doc" : {"follower\_num" : 10} }

{ "update": { "\_id": "3"} }

{ "doc" : {"follower\_num" : 25} }

{ "update": { "\_id": "4"} }

{ "doc" : {"follower\_num" : 3} }

{ "update": { "\_id": "5"} }

{ "doc" : {"follower\_num" : 60} }

将对帖子搜索得到的分数，跟follower\_num进行运算，由follower\_num在一定程度上增强帖子的分数

看帖子的人越多，那么帖子的分数就越高

GET /forum/article/\_search

{

"query": {

**"function\_score"**: {

"query": {

"multi\_match": {

"query": "java spark",

"fields": ["tile", "content"]

}

},

"field\_value\_factor": {

"field": "follower\_num",

**"modifier": "log1p",**

"factor": 0.5

},

"boost\_mode": "sum",

"max\_boost": 2

}

}

}

如果只有field，那么会将每个doc的分数都乘以follower\_num，如果有的doc follower是0，那么分数就会变为0，效果很不好。因此一般会加个log1p函数，公式会变为，new\_score = old\_score \* log(1 + number\_of\_votes)，这样出来的分数会比较合理

再加个factor，可以进一步影响分数，new\_score = old\_score \* log(1 + factor \* number\_of\_votes)

boost\_mode，可以决定分数与指定字段的值如何计算，multiply，sum，min，max，replace

max\_boost，限制计算出来的分数不要超过max\_boost指定的值

### 实战-误拼写时的 fuzzy 模糊搜索技术

搜索的时候，可能输入的搜索文本会出现误拼写的情况

doc1: hello world

doc2: hello java

搜索：hallo world

fuzzy搜索技术 --> 自动将拼写错误的搜索文本，进行纠正，纠正以后去尝试匹配索引中的数据

添加测试数据：

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": 1 }}

{ "text": "Surprise me!"}

{ "index": { "\_id": 2 }}

{ "text": "That was surprising."}

{ "index": { "\_id": 3 }}

{ "text": "I wasn't surprised."}

执行 fuzzy 搜索：

GET /my\_index/my\_type/\_search

{

"query": {

"fuzzy": {

"text": {

"value": "surprize",

"fuzziness": 2

}

}

}

}

surprize --> 拼写错误 --> surprise --> s -> z

surprize --> surprise -> z -> s，纠正一个字母，就可以匹配上，所以在fuziness指定的2范围内

surprize --> surprised -> z -> s，末尾加个d，纠正了2次，也可以匹配上，在fuziness指定的2范围内

surprize --> surprising -> z -> s，去掉e，ing，3次，总共要5次，才可以匹配上，始终纠正不了

fuzzy搜索以后，会自动尝试将你的搜索文本进行纠错，然后去跟文本进行匹配

fuzziness，你的搜索文本最多可以纠正几个字母去跟你的数据进行匹配，默认如果不设置，就是2

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"text": {

"query": "SURPIZE ME",

"fuzziness": "AUTO",

"operator": "and"

}

}

}

}

## 掌握 IK 中文分词

### 上机动手实战 IK 中文分词器的安装和使用

之前大家会发现，我们全部是用英文在玩儿。。。好玩儿不好玩儿。。。不好玩儿

中国人，其实我们用来进行搜索的，绝大多数，都是中文应用，很少做英文的

standard：没有办法对中文进行合理分词的，只是将每个中文字符一个一个的切割开来，比如说中国人 --> 中 国 人

英语的也要学：所以说，我们利用核心知识篇的相关的知识，来把es这种英文原生的搜索引擎，先学一下; 因为有些知识点，可能用英文讲更靠谱，因为比如说analyzed，palyed，students --> stemmer，analyze，play，student。有些知识点，仅仅适用于英文，不太适用于中文

从这一讲开始，大家就会觉得很爽，因为全部都是我们熟悉的中文了，没有英文了，高阶知识点，搜索，聚合，全部是中文了

在搜索引擎领域，比较成熟和流行的，就是ik分词器

中国人很喜欢吃油条

standard：中 国 人 很 喜 欢 吃 油 条

ik：中国人 很 喜欢 吃 油条

1、在elasticsearch中安装ik中文分词器

（1）git clone https://github.com/medcl/elasticsearch-analysis-ik

（2）git checkout tags/v5.2.0

（3）mvn package

（4）将target/releases/elasticsearch-analysis-ik-5.2.0.zip拷贝到es/plugins/ik目录下

（5）在es/plugins/ik下对elasticsearch-analysis-ik-5.2.0.zip进行解压缩

（6）重启es

2、ik分词器基础知识

两种analyzer，你根据自己的需要自己选吧，但是一般是选用ik\_max\_word

\*\* ik\_max\_word: 会将文本做最细粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,中华人民,中华,华人,人民共和国,人民,人,民,共和国,共和,和,国国,国歌”，会穷尽各种可能的组合；

\*\* ik\_smart: 会做最粗粒度的拆分，比如会将“中华人民共和国国歌”拆分为“中华人民共和国,国歌”。

共和国 --> 中华人民共和国和国歌，搜到吗？？？？

3、ik分词器的使用

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

POST /my\_index/my\_type/\_bulk

{ "index": { "\_id": "1"} }

{ "text": "男子偷上万元发红包求交女友 被抓获时仍然单身" }

{ "index": { "\_id": "2"} }

{ "text": "16岁少女为结婚“变”22岁 7年后想离婚被法院拒绝" }

{ "index": { "\_id": "3"} }

{ "text": "深圳女孩骑车逆行撞奔驰 遭索赔被吓哭(图)" }

{ "index": { "\_id": "4"} }

{ "text": "女人对护肤品比对男票好？网友神怼" }

{ "index": { "\_id": "5"} }

{ "text": "为什么国内的街道招牌用的都是红黄配？" }

GET /my\_index/\_analyze

{

"text": "男子偷上万元发红包求交女友 被抓获时仍然单身",

"analyzer": "ik\_max\_word"

}

GET /my\_index/my\_type/\_search

{

"query": {

"match": {

"text": "16岁少女结婚好还是单身好？"

}

}

}

### IK 分词器配置文件讲解以及自定义词库实战

1、ik配置文件

ik配置文件地址：es/plugins/ik/config目录

IKAnalyzer.cfg.xml：用来配置自定义词库

main.dic：ik原生内置的中文词库，总共有27万多条，只要是这些单词，都会被分在一起

quantifier.dic：放了一些单位相关的词

suffix.dic：放了一些后缀

surname.dic：中国的姓氏

stopword.dic：英文停用词

ik原生最重要的两个配置文件

main.dic：包含了原生的中文词语，会按照这个里面的词语去分词

stopword.dic：包含了英文的停用词

停用词，stopword

a the and at but

一般，像停用词，会在分词的时候，直接被干掉，不会建立在倒排索引中

2、自定义词库

（1）自己建立词库：每年都会涌现一些特殊的流行词，网红，蓝瘦香菇，喊麦，鬼畜，一般不会在ik的原生词典里

自己补充自己的最新的词语，到ik的词库里面去

IKAnalyzer.cfg.xml：ext\_dict，custom/mydict.dic

补充自己的词语，然后需要重启es，才能生效

（2）自己建立停用词库：比如了，的，啥，么，我们可能并不想去建立索引，让人家搜索

custom/ext\_stopword.dic，已经有了常用的中文停用词，可以补充自己的停用词，然后重启es

### 修改 IK 分词器源码来基于 mysql 热更新词库

背景介绍：每次都是在es的扩展词典中，手动添加新词语，很坑

（1）每次添加完，都要重启es才能生效，非常麻烦

（2）es是分布式的，可能有数百个节点，你不能每次都一个一个节点上面去修改

需要的场景：es不停机，直接我们在外部某个地方添加新的词语，es中立即热加载到这些新词语

热更新的方案

（1）修改ik分词器源码，然后手动支持从mysql中每隔一定时间，自动加载新的词库

（2）基于ik分词器原生支持的热更新方案，部署一个web服务器，提供一个http接口，通过modified和tag两个http响应头，来提供词语的热更新

用第一种方案，第二种，ik git社区官方都不建议采用，觉得不太稳定

1、下载源码

https://github.com/medcl/elasticsearch-analysis-ik/tree/v5.2.0

ik分词器，是个标准的java maven工程，直接导入eclipse就可以看到源码

2、修改源码

Dictionary类，169行：Dictionary单例类的初始化方法，在这里需要创建一个我们自定义的线程，并且启动它

HotDictReloadThread类：就是死循环，不断调用Dictionary.getSingleton().reLoadMainDict()，去重新加载词典

**Dictionary类，389行：this.loadMySQLExtDict();**

Dictionary类，683行：this.loadMySQLStopwordDict();

3、mvn package打包代码

target\releases\elasticsearch-analysis-ik-5.2.0.zip

4、解压缩ik压缩包

将mysql驱动jar，放入ik的目录下

5、修改jdbc相关配置

6、重启es

观察日志，日志中就会显示我们打印的那些东西，比如加载了什么配置，加载了什么词语，什么停用词

7、在mysql中添加词库与停用词

8、分词实验，验证热更新生效

## 深入聚合数据分析

### 概念-bucket与 metric 两个核心概念的讲解

课程大纲

1、文本编辑器介绍

（1）windows操作系统，原生的txt文本编辑器，一些json格式，不太方便去调整

（2）notepad++，功能不是太丰富

（3）sublime，整个功能也比较丰富，比较好，自己可以上网去下载，官网，免费的

2、两个核心概念：bucket和metric

**\*\* bucket：一个数据分组**

city name

北京 小李

北京 小王

上海 小张

上海 小丽

上海 小陈

基于city划分buckets

划分出来两个bucket，一个是北京bucket，一个是上海bucket

北京bucket：包含了2个人，小李，小王

上海bucket：包含了3个人，小张，小丽，小陈

按照某个字段进行bucket划分，那个字段的值相同的那些数据，就会被划分到一个bucket中

有一些mysql的sql知识的话，聚合，首先第一步就是分组，对每个组内的数据进行聚合分析，分组，就是我们的bucket

**\*\* metric：对一个数据分组执行的统计**

当我们有了一堆bucket之后，就可以对每个bucket中的数据进行聚合分词了，比如说计算一个bucket内所有数据的数量，或者计算一个bucket内所有数据的平均值，最大值，最小值

metric，就是对一个bucket执行的某种聚合分析的操作，比如说求平均值，求最大值，求最小值

select count(\*)

from access\_log

group by user\_id

bucket：group by user\_id --> 那些user\_id相同的数据，就会被划分到一个bucket中

metric：count(\*)，对每个user\_id bucket中所有的数据，计算一个数量

### 实战-家电卖场案例以及统计哪种颜色电视销量最高

1、家电卖场案例背景

**以一个家电卖场中的电视销售数据为背景**，来对各种品牌，各种颜色的电视的销量和销售额，进行各种各样角度的分析

创建 tvs 的索引，价格(price),颜色(color),品牌(brand),销售日期(sold\_date)

PUT /tvs

{

"mappings": {

"sales": {

"properties": {

"price": {

"type": "long"

},

"color": {

"type": "keyword"

},

"brand": {

"type": "keyword"

},

"sold\_date": {

"type": "date"

}

}

}

}

}

创建案例数据：

POST /tvs/sales/\_bulk

{ "index": {}}

{ "price" : 1000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-10-28" }

{ "index": {}}

{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-11-05" }

{ "index": {}}

{ "price" : 3000, "color" : "绿色", "brand" : "小米", "sold\_date" : "2016-05-18" }

{ "index": {}}

{ "price" : 1500, "color" : "蓝色", "brand" : "TCL", "sold\_date" : "2016-07-02" }

{ "index": {}}

{ "price" : 1200, "color" : "绿色", "brand" : "TCL", "sold\_date" : "2016-08-19" }

{ "index": {}}

{ "price" : 2000, "color" : "红色", "brand" : "长虹", "sold\_date" : "2016-11-05" }

{ "index": {}}

{ "price" : 8000, "color" : "红色", "brand" : "三星", "sold\_date" : "2017-01-01" }

{ "index": {}}

{ "price" : 2500, "color" : "蓝色", "brand" : "小米", "sold\_date" : "2017-02-12" }

2、统计哪种颜色的电视销量最高

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"popular\_colors" : {

"terms" : {

"field" : "color"

}

}

}

}

size：只获取聚合结果，而不要执行聚合的原始数据

aggs：固定语法，要对一份数据执行分组聚合操作

popular\_colors：就是对每个aggs，都要起一个名字，这个名字是随机的，你随便取什么都可以

terms：根据字段的值进行分组

field：根据指定的字段的值进行分组

结果：

|  |
| --- |
| {  "took": 61,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "popular\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "红色",  "doc\_count": 4  },  {  "key": "绿色",  "doc\_count": 2  },  {  "key": "蓝色",  "doc\_count": 2  }  ]  }  }  } |

hits.hits：我们指定了size是0，所以hits.hits就是空的，否则会把执行聚合的那些原始数据给你返回回来

aggregations：聚合结果

popular\_color：我们指定的某个聚合的名称

buckets：根据我们指定的field划分出的buckets

key：每个bucket对应的那个值

doc\_count：这个bucket分组内，有多少个数据

数量，其实就是这种颜色的销量

每种颜色对应的bucket中的数据的

默认的排序规则：按照doc\_count降序排序

### 实战-bucket+metric:统计每种颜色电视平均价格

select average(sum(price)) as price, color

from tvs

group by color

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"colors": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

按照color去分bucket，可以拿到每个color bucket中的数量，这个仅仅只是一个bucket操作，doc\_count其实只是es的bucket操作默认执行的一个内置metric

这一讲，就是除了bucket操作，分组，还要对每个bucket执行一个metric聚合统计操作

在一个aggs执行的bucket操作（terms），平级的json结构下，再加一个aggs，这个第二个aggs内部，同样取个名字，执行一个metric操作，avg，对之前的每个bucket中的数据的指定的field，price field，求一个平均值

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

就是一个metric，就是一个对一个bucket分组操作之后，对每个bucket都要执行的一个metric

第一个metric，avg，求指定字段的平均值

{

"took": 28,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 8,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_color": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "红色",

"doc\_count": 4,

"avg\_price": {

"value": 3250

}

},

{

"key": "绿色",

"doc\_count": 2,

"avg\_price": {

"value": 2100

}

},

{

"key": "蓝色",

"doc\_count": 2,

"avg\_price": {

"value": 2000

}

}

]

}

}

}

buckets，除了key和doc\_count

avg\_price：我们自己取的metric aggs的名字

value：我们的metric计算的结果，每个bucket中的数据的price字段求平均值后的结果

select avg(price)

from tvs.sales

group by color

### 实战-bucket 嵌套实现颜色+品牌的多层下钻分析

从颜色到品牌进行下钻分析，每种颜色的平均价格，以及找到每种颜色每个品牌的平均价格

我们可以进行多层次的下钻

比如说，现在红色的电视有4台，同时这4台电视中，有3台是属于长虹的，1台是属于小米的

红色电视中的3台长虹的平均价格是多少？

红色电视中的1台小米的平均价格是多少？

下钻的意思是，已经分了一个组了，比如说颜色的分组，然后还要继续对这个分组内的数据，再分组，比如一个颜色内，还可以分成多个不同的品牌的组，最后对每个最小粒度的分组执行聚合分析操作，这就叫做下钻分析

es，下钻分析，就要对bucket进行多层嵌套，多次分组

按照多个维度（颜色+品牌）多层下钻分析，而且学会了每个下钻维度（颜色，颜色+品牌），都可以对每个维度分别执行一次metric聚合操作

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"color\_avg\_price": {

"avg": {

"field": "price"

}

},

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"brand\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 8,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "红色",  "doc\_count": 4,  "color\_avg\_price": {  "value": 3250  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "长虹",  "doc\_count": 3,  "brand\_avg\_price": {  "value": 1666.6666666666667  }  },  {  "key": "三星",  "doc\_count": 1,  "brand\_avg\_price": {  "value": 8000  }  }  ]  }  },  {  "key": "绿色",  "doc\_count": 2,  "color\_avg\_price": {  "value": 2100  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "TCL",  "doc\_count": 1,  "brand\_avg\_price": {  "value": 1200  }  },  {  "key": "小米",  "doc\_count": 1,  "brand\_avg\_price": {  "value": 3000  }  }  ]  }  },  {  "key": "蓝色",  "doc\_count": 2,  "color\_avg\_price": {  "value": 2000  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "TCL",  "doc\_count": 1,  "brand\_avg\_price": {  "value": 1500  }  },  {  "key": "小米",  "doc\_count": 1,  "brand\_avg\_price": {  "value": 2500  }  }  ]  }  }  ]  }  }  } |

### 实战-max,min入门：统计每种颜色电视最大最小价格

之前我们学习了如下的 metric：

\*\* count：bucket，terms，自动就会有一个doc\_count，就相当于是count

\*\* avg：avg aggs，求平均值

现在学习下面的 metric：

\*\* max：求一个bucket内，指定field值最大的那个数据

\*\* min：求一个bucket内，指定field值最小的那个数据

\*\* sum：求一个bucket内，指定field值的总和

一般来说，90%的常见的数据分析的操作，metric，无非就是count，avg，max，min，sum

m

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": { "avg": { "field": "price" } },

"min\_price" : { "min": { "field": "price"} },

"max\_price" : { "max": { "field": "price"} },

"sum\_price" : { "sum": { "field": "price" } }

}

}

}

}

求总和，就可以拿到一个颜色下的所有电视的销售总额

|  |
| --- |
| {  "took": 16,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "红色",  "doc\_count": 4,  "max\_price": {  "value": 8000  },  "min\_price": {  "value": 1000  },  "avg\_price": {  "value": 3250  },  "sum\_price": {  "value": 13000  }  },  {  "key": "绿色",  "doc\_count": 2,  "max\_price": {  "value": 3000  },  "min\_price": {  "value":  }, 1200  "avg\_price": {  "value": 2100  },  "sum\_price": {  "value": 4200  }  },  {  "key": "蓝色",  "doc\_count": 2,  "max\_price": {  "value": 2500  },  "min\_price": {  "value": 1500  },  "avg\_price": {  "value": 2000  },  "sum\_price": {  "value": 4000  }  }  ]  }  }  } |

### 实战-histogram按价格区间统计电视销量和销售额

SELECT sum(price) FROM tvs GROUP BY date\_histogram(field=' price,'interval'='2000')

histogram：类似于terms，也是进行bucket分组操作，接收一个field，按照这个field的值的各个范围区间，进行bucket分组操作

"histogram":{

"field": "price",

"interval": 2000

},

interval：2000，划分范围，0~2000，2000~4000，4000~6000，6000~8000，8000~10000，buckets

去根据price的值，比如2500，看落在哪个区间内，比如2000~4000，此时就会将这条数据放入2000~4000对应的那个bucket中

bucket划分的方法：terms，将field值相同的数据划分到一个bucket中

bucket有了之后，一样的，去对每个bucket执行avg，count，sum，max，min，等各种metric操作，聚合分析

GET /tvs/sales/\_search

{

"size" : 0,

"aggs":{

"group\_by\_price":{

"histogram":{

"field": "price",

"interval": 2000

},

"aggs":{

"revenue": {

"sum": {

"field" : "price"

}

}

}

}

}

}

|  |
| --- |
| {  "took": 13,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_price": {  "buckets": [  {  "key": 0,  "doc\_count": 3,  "sum\_price": {  "value": 3700  }  },  {  "key": 2000,  "doc\_count": 4,  "sum\_price": {  "value": 9500  }  },  {  "key": 4000,  "doc\_count": 0,  "sum\_price": {  "value": 0  }  },  {  "key": 6000,  "doc\_count: {  "value":": 0,  "sum\_price" 0  }  },  {  "key": 8000,  "doc\_count": 1,  "sum\_price": {  "value": 8000  }  }  ]  }  }  } |

### 实战-date\_histogram之统计每月电视销量

bucket，分组操作，histogram，按照某个值指定的interval，划分一个一个的bucket

date histogram，按照我们指定的某个date类型的日期field，以及日期interval，按照一定的日期间隔，去划分bucket

date interval = 1m，

2017-01-01~2017-01-31，就是一个bucket

2017-02-01~2017-02-28，就是一个bucket

然后会去扫描每个数据的date field，判断date落在哪个bucket中，就将其放入那个bucket

2017-01-05，就将其放入2017-01-01~2017-01-31，就是一个bucket

min\_doc\_count：即使某个日期interval，2017-01-01~2017-01-31中，一条数据都没有，那么这个区间也是要返回的，不然默认是会过滤掉这个区间的

extended\_bounds，min，max：划分bucket的时候，会限定在这个起始日期，和截止日期内

GET /tvs/sales/\_search

{

"size" : 0,

"aggs": {

"group\_by\_sold\_date": {

**"date\_histogram":** {

"field": "sold\_date",

"interval": "month",

"format": "yyyy-MM-dd",

"min\_doc\_count" : 0,

"extended\_bounds" : {

"min" : "2016-01-01",

"max" : "2017-12-31"

}

}

}

}

}

|  |
| --- |
| {  "took": 16,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_sold\_date": {  "buckets": [  {  "key\_as\_string": "2016-01-01",  "key": 1451606400000,  "doc\_count": 0  },  {  "key\_as\_string": "2016-02-01",  "key": 1454284800000,  "doc\_count": 0  },  {  "key\_as\_string": "2016-03-01",  "key": 1456790400000,  "doc\_count": 0  },  {  "key\_as\_string": "2016-04-01",  "key": 1459468800000,  "doc\_count": 0  },  {  "key\_as\_string": "2016-05-01",  "key": 1462060800000,  "doc\_count": 1  },  {  "key\_as\_string": "2016-06-01",  "key": 1464739200000,  "doc\_count": 0  },  {  "key\_as\_string": "2016-07-01",  "key": 1467331200000,  "doc\_count": 1  },  {  "key\_as\_strin  "key\_as\_string": "2016-09-01",  "key": 1472688000000,  "doc\_count": 0  },g": "2016-08-01",  "key": 1470009600000,  "doc\_count": 1  },  {  {  "key\_as\_string": "2016-10-01",  "key": 1475280000000,  "doc\_count": 1  },  {  "key\_as\_string": "2016-11-01",  "key": 1477958400000,  "doc\_count": 2  },  {  "key\_as\_string": "2016-12-01",  "key": 1480550400000,  "doc\_count": 0  },  {  "key\_as\_string": "2017-01-01",  "key": 1483228800000,  "doc\_count": 1  },  {  "key\_as\_string": "2017-02-01",  "key": 1485907200000,  "doc\_count": 1  }  ]  }  }  } |

### 实战-下钻分析之统计每季度每个品牌的销售额

quarter 表示按季度区分

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_sold\_date": {

"date\_histogram": {

"field": "sold\_date",

"interval": "quarter",

"format": "yyyy-MM-dd",

"min\_doc\_count": 0,

"extended\_bounds": {

"min": "2016-01-01",

"max": "2017-12-31"

}

},

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand"

},

"aggs": {

"sum\_price": {

"sum": {

"field": "price"

}

}

}

},

"total\_sum\_price": {

"sum": {

"field": "price"

}

}

}

}

}

}

|  |
| --- |
| {  "took": 10,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_sold\_date": {  "buckets": [  {  "key\_as\_string": "2016-01-01",  "key": 1451606400000,  "doc\_count": 0,  "total\_sum\_price": {  "value": 0  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": []  }  },  {  "key\_as\_string": "2016-04-01",  "key": 1459468800000,  "doc\_count": 1,  "total\_sum\_price": {  "value": 3000  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "小米",  "doc\_count": 1,  "sum\_price": {  "value": 3000  }  }  ]  }  },  {  "key\_as\_string": "2016-07-01",  "key": 1467331200000,  "doc\_count": 2,  "total\_sum\_price": {  "value": 2700  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "TCL",  "doc\_count": 2,  "sum\_price": {  "value": 2700  }  }  ]  }  },  {  "key\_as\_string": "2016-10-01",  "key": 1475280000000,  "doc\_count": 3,  "total\_sum\_price": {  "value": 5000  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "长虹",  "doc\_count": 3,  "sum\_price": {  "value": 5000  }  }  ]  }  },  {  "key\_as\_string": "2017-01-01",  "key": 1483228800000,  "doc\_count": 2,  "total\_sum\_price": {  "value": 10500  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "三星",  "doc\_count": 1,  "sum\_price": {  "value": 8000  }  },  {  "key": "小米",  "doc\_count": 1,  "sum\_price": {  "value": 2500  }  }  ]  }  },  {  "key\_as\_string": "2017-04-01",  "key": 1491004800000,  "doc\_count": 0,  "total\_sum\_price": {  "value": 0  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": []  }  },  {  "key\_as\_string": "2017-07-01",  "key": 1498867200000,  "doc\_count": 0,  "total\_sum\_price": {  "value": 0  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": []  }  },  {  "key\_as\_string": "2017-10-01",  "key": 1506816000000,  "doc\_count": 0,  "total\_sum\_price": {  "value": 0  },  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": []  }  }  ]  }  }  } |

### 实战-搜索+聚合：统计指定品牌下每个颜色的销量

实际上来说，我们之前学习的搜索相关的知识，完全可以和聚合组合起来使用

select count(\*)

from tvs.sales

where brand like "%长%"

group by price

es aggregation，scope，任何的聚合，都必须在搜索出来的结果数据中之行，搜索结果，就是聚合分析操作的scope

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "小米"

}

}

},

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

}

}

}

}

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "绿色",  "doc\_count": 1  },  {  "key": "蓝色",  "doc\_count": 1  }  ]  }  }  } |

### 实战-global bucket：单个品牌与所有品牌销售量对比

aggregation，scope，一个聚合操作，必须在query的搜索结果范围内执行

出来两个结果，一个结果，是基于query搜索结果来聚合的; 一个结果，是对所有数据执行聚合的

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "长虹"

}

}

},

"aggs": {

"single\_brand\_avg\_price": {

"avg": {

"field": "price"

}

},

"all": {

**"global": {},**

"aggs": {

"all\_brand\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

global：就是global bucket，就是将所有数据纳入聚合的scope，而不管之前的query

|  |
| --- |
| {  "took": 4,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 0,  "hits": []  },  "aggregations": {  "all": {  "doc\_count": 8,  "all\_brand\_avg\_price": {  "value": 2650  }  },  "single\_brand\_avg\_price": {  "value": 1666.6666666666667  }  }  } |

single\_brand\_avg\_price：就是针对query搜索结果，执行的，拿到的，就是长虹品牌的平均价格

all.all\_brand\_avg\_price：拿到所有品牌的平均价格

### 实战-过滤+聚合：统计价格大于 1200 的电视平均价格

搜索+聚合

过滤+聚合

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"constant\_score": {

"filter": {

"range": {

"price": {

"gte": 1200

}

}

}

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

|  |
| --- |
| {  "took": 41,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 7,  "max\_score": 0,  "hits": []  },  "aggregations": {  "avg\_price": {  "value": 2885.714285714286  }  }  } |

### 实战-bucket filter：统计品牌最近一个月的平均价格

GET /tvs/sales/\_search

{

"size": 0,

"query": {

"term": {

"brand": {

"value": "长虹"

}

}

},

"aggs": {

"recent\_150d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-150d"

}

}

},

"aggs": {

"recent\_150d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

},

"recent\_140d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-140d"

}

}

},

"aggs": {

"recent\_140d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

},

"recent\_130d": {

"filter": {

"range": {

"sold\_date": {

"gte": "now-130d"

}

}

},

"aggs": {

"recent\_130d\_avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

aggs.filter，针对的是聚合去做的

如果放query里面的filter，是全局的，会对所有的数据都有影响

但是，如果，比如说，你要统计，长虹电视，最近1个月的平均值; 最近3个月的平均值; 最近6个月的平均值

bucket filter：对不同的bucket下的aggs，进行filter

### 实战-排序：按每种颜色的平均销售额降序排序

之前的话，排序，是按照每个bucket的doc\_count降序来排的

但是假如说，我们现在统计出来每个颜色的电视的销售额，需要按照销售额降序排序？？？？

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color",

"order": {

"avg\_price": "asc"

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "红色",  "doc\_count": 4,  "avg\_price": {  "value": 3250  }  },  {  "key": "绿色",  "doc\_count": 2,  "avg\_price": {  "value": 2100  }  },  {  "key": "蓝色",  "doc\_count": 2,  "avg\_price": {  "value": 2000  }  }  ]  }  }  } |

### 实战-颜色+品牌下钻分析是按最深 metric 进行排序

GET /tvs/sales/\_search

{

"size": 0,

"aggs": {

"group\_by\_color": {

"terms": {

"field": "color"

},

"aggs": {

"group\_by\_brand": {

"terms": {

"field": "brand",

"order": {

"avg\_price": "desc"

}

},

"aggs": {

"avg\_price": {

"avg": {

"field": "price"

}

}

}

}

}

}

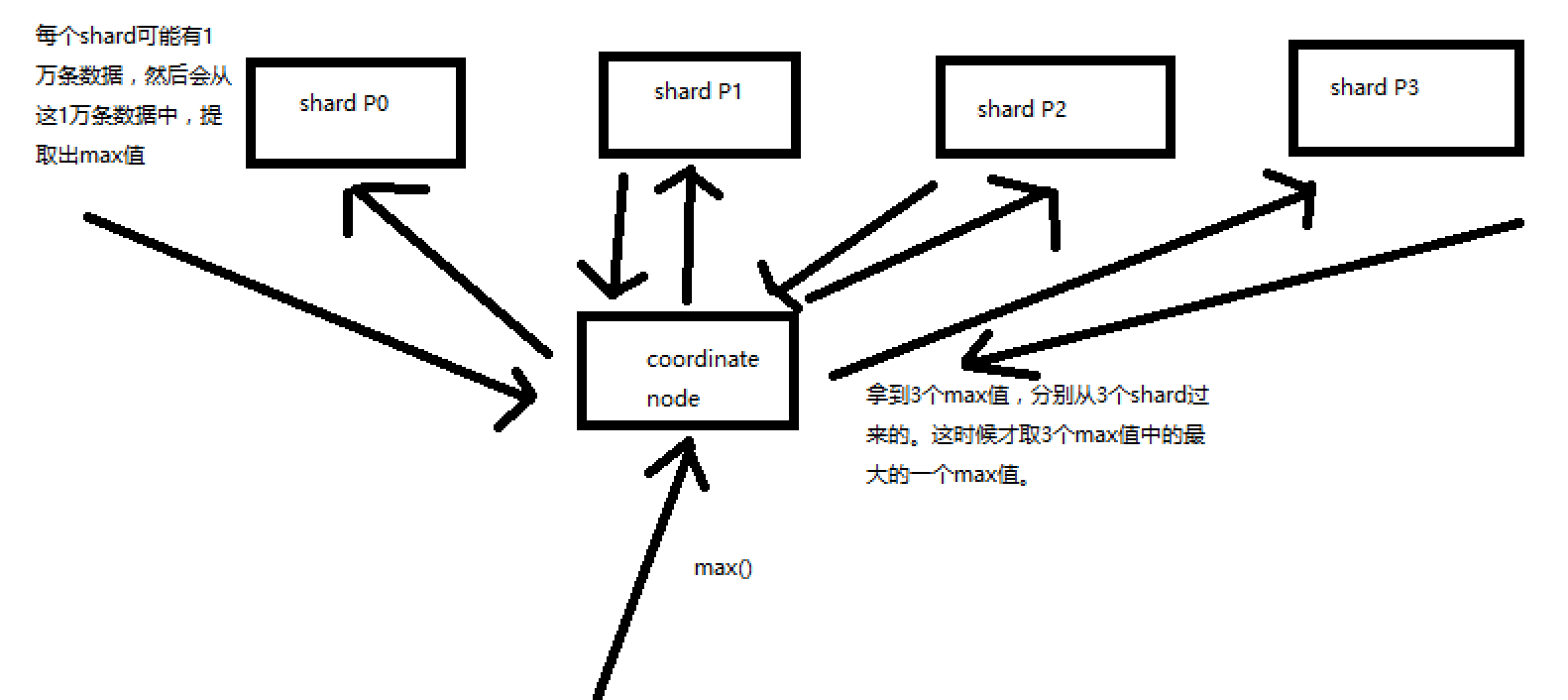
}

}

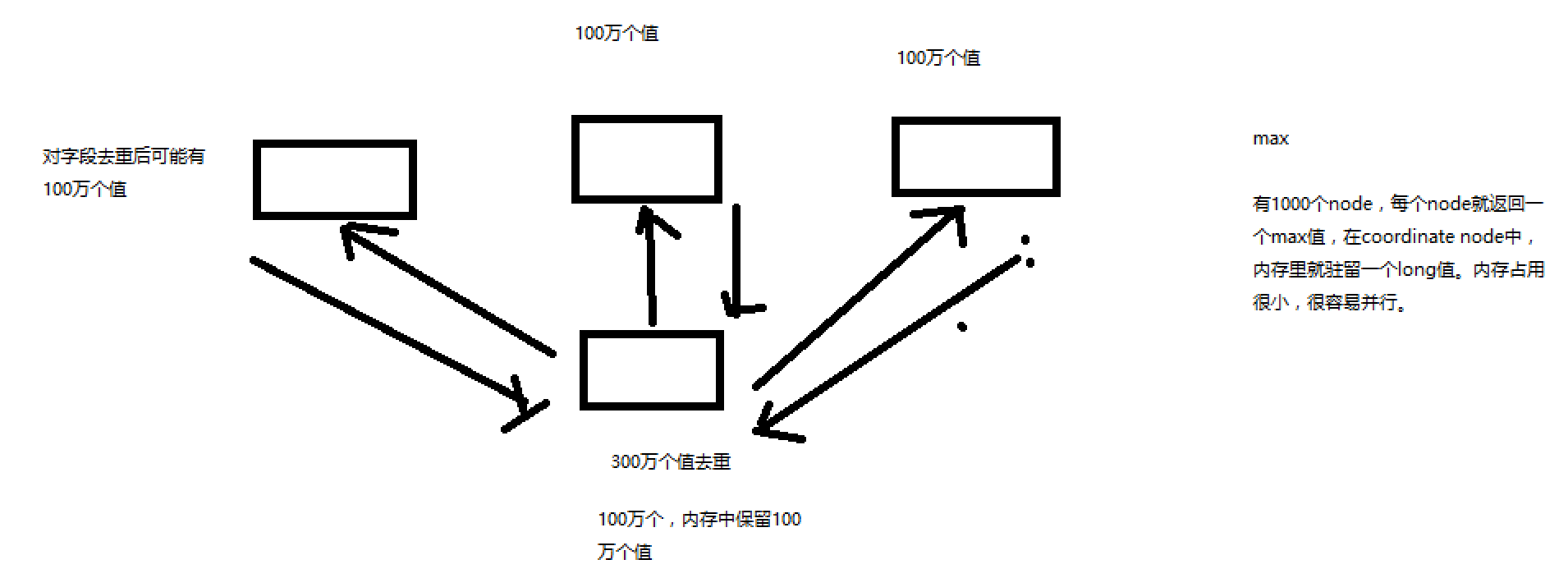
|  |
| --- |
| {  "took": 4,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_color": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "红色",  "doc\_count": 4,  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "三星",  "doc\_count": 1,  "avg\_price": {  "value": 8000  }  },  {  "key": "长虹",  "doc\_count": 3,  "avg\_price": {  "value": 1666.6666666666667  }  }  ]  }  },  {  "key": "绿色",  "doc\_count": 2,  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "小米",  "doc\_count": 1,  "avg\_price": {  "value": 3000  }  },  {  "key": "TCL",  "doc\_count": 1,  "avg\_price": {  "value": 1200  }  }  ]  }  },  {  "key": "蓝色",  "doc\_count": 2,  "group\_by\_brand": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "小米",  "doc\_count": 1,  "avg\_price": {  "value": 2500  }  },  {  "key": "TCL",  "doc\_count": 1,  "avg\_price": {  "value": 1500  }  }  ]  }  }  ]  }  }  } |

### 原理-易并行聚合算法，三角选择原则，近似聚合算法

1、画图讲解易并行聚合算法：max

有些聚合分析的算法，是很容易就可以并行的，比如说max

有些聚合分析的算法，是不好并行的，比如说，count(distinct)，并不是说，在每个node上，直接就出一些distinct value，就可以的，因为数据可能会很多



es会采取近似聚合的方式，就是采用在每个node上进行近估计的方式，得到最终的结论，cuont(distcint)，100万，1050万/95万 --> 5%左右的错误率

近似估计后的结果，不完全准确，但是速度会很快，一般会达到完全精准的算法的性能的数十倍

2、三角选择原则

精准+实时+大数据 --> 选择2个

（1）精准+实时: 没有大数据，数据量很小，那么一般就是单击跑，随便你则么玩儿就可以

（2）精准+大数据：hadoop，批处理，非实时，可以处理海量数据，保证精准，可能会跑几个小时

（3）大数据+实时：es，不精准，近似估计，可能会有百分之几的错误率

3、近似聚合算法

如果采取近似估计的算法：延时在100ms左右，0.5%错误

如果采取100%精准的算法：延时一般在5s~几十s，甚至几十分钟，几小时， 0%错误

### 实战-cardinality 去重算法以及每月销售额品牌数量统计

es，去重，**cardinality metric**，对每个bucket中的指定的field进行去重，取去重后的count，类似于count(distcint)

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"group\_by\_sold\_date": {

"date\_histogram": {

"field": "sold\_date",

"interval": "month"

},

"aggs": {

"distinct\_colors": {

**"cardinality"**: {

"field": "brand"

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 70,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 8,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_sold\_date": {  "buckets": [  {  "key\_as\_string": "2016-05-01T00:00:00.000Z",  "key": 1462060800000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2016-06-01T00:00:00.000Z",  "key": 1464739200000,  "doc\_count": 0,  "distinct\_brand\_cnt": {  "value": 0  }  },  {  "key\_as\_string": "2016-07-01T00:00:00.000Z",  "key": 1467331200000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2016-08-01T00:00:00.000Z",  "key": 1470009600000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2016-09-01T00:00:00.000Z",  "key": 1472688000000,  "doc\_count": 0,  "distinct\_brand\_cnt": {  "value": 0  }  },  {  "key\_as\_string": "2016-10-01T00:00:00.000Z",  "key": 1475280000000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2016-11-01T00:00:00.000Z",  "key": 1477958400000,  "doc\_count": 2,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2016-12-01T00:00:00.000Z",  "key": 1480550400000,  "doc\_count": 0,  "distinct\_brand\_cnt": {  "value": 0  }  },  {  "key\_as\_string": "2017-01-01T00:00:00.000Z",  "key": 1483228800000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  },  {  "key\_as\_string": "2017-02-01T00:00:00.000Z",  "key": 1485907200000,  "doc\_count": 1,  "distinct\_brand\_cnt": {  "value": 1  }  }  ]  }  }  } |

### 实战-cardinality 算法之优化内存开销以及 HLL 算法

cardinality，count(distinct)，5%的错误率，性能在100ms左右

1、precision\_threshold优化准确率和内存开销

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"distinct\_brand" : {

"cardinality" : {

"field" : "brand",

"precision\_threshold" : 100

}

}

}

}

brand去重，如果brand的unique value，在100个以内，小米，长虹，三星，TCL，HTL。。。

在多少个unique value以内，cardinality，几乎保证100%准确

cardinality算法，会占用precision\_threshold \* 8 byte 内存消耗，100 \* 8 = 800个字节

占用内存很小。。。而且unique value如果的确在值以内，那么可以确保100%准确

100，数百万的unique value，错误率在5%以内

precision\_threshold，值设置的越大，占用内存越大，1000 \* 8 = 8000 / 1000 = 8KB，可以确保更多unique value的场景下，100%的准确

field，去重，count，这时候，unique value，10000，precision\_threshold=10000，10000 \* 8 = 80000个byte，80KB

2、HyperLogLog++ (HLL)算法性能优化

cardinality底层算法：HLL算法，HLL算法的性能

会对所有的uqniue value取hash值，通过hash值近似去求distcint count，误差

默认情况下，发送一个cardinality请求的时候，会动态地对所有的field value，取hash值; 将取hash值的操作，前移到建立索引的时候

PUT /tvs/

{

"mappings": {

"sales": {

"properties": {

"brand": {

"type": "text",

"fields": {

"hash": {

"type": "murmur3"

}

}

}

}

}

}

}

GET /tvs/sales/\_search

{

"size" : 0,

"aggs" : {

"distinct\_brand" : {

"cardinality" : {

"field" : "brand.hash",

"precision\_threshold" : 100

}

}

}

}

### 实战-perentilies 百分比算法以及网站访问延时统计

百分位数聚合，统计学里面的概念。可以参考统计学里面的计算方法。

需求：比如有一个网站，记录下了每次请求的访问的耗时，需要统计tp50，tp90，tp99

tp50：50%的请求的耗时最长在多长时间

tp90：90%的请求的耗时最长在多长时间

tp99：99%的请求的耗时最长在多长时间

PUT /website

{

"mappings": {

"logs": {

"properties": {

"latency": {

"type": "long"

},

"province": {

"type": "keyword"

},

"timestamp": {

"type": "date"

}

}

}

}

}

POST /website/logs/\_bulk

{ "index": {}}

{ "latency" : 105, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 83, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 92, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 112, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 68, "province" : "江苏", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 76, "province" : "江苏", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 101, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 275, "province" : "新疆", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 166, "province" : "新疆", "timestamp" : "2016-10-29" }

{ "index": {}}

{ "latency" : 654, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 389, "province" : "新疆", "timestamp" : "2016-10-28" }

{ "index": {}}

{ "latency" : 302, "province" : "新疆", "timestamp" : "2016-10-29" }

pencentiles

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"latency\_percentiles": {

**"percentiles":** {

"field": "latency",

"percents": [

50,

95,

99

]

}

},

"latency\_avg": {

"avg": {

"field": "latency"

}

}

}

}

|  |
| --- |
| {  "took": 31,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 12,  "max\_score": 0,  "hits": []  },  "aggregations": {  "latency\_avg": {  "value": 201.91666666666666  },  "latency\_percentiles": {  "values": {  "50.0": 108.5,  "95.0": 508.24999999999983,  "99.0": 624.8500000000001  }  }  }  } |

50%的请求，数值的最大的值是多少，不是完全准确的

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"group\_by\_province": {

"terms": {

"field": "province"

},

"aggs": {

"latency\_percentiles": {

"percentiles": {

"field": "latency",

"percents": [

50,

95,

99

]

}

},

"latency\_avg": {

"avg": {

"field": "latency"

}

}

}

}

}

}

|  |
| --- |
| {  "took": 33,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 12,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_province": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "新疆",  "doc\_count": 6,  "latency\_avg": {  "value": 314.5  },  "latency\_percentiles": {  "values": {  "50.0": 288.5,  "95.0": 587.75,  "99.0": 640.75  }  }  },  {  "key": "江苏",  "doc\_count": 6,  "latency\_avg": {  "value": 89.33333333333333  },  "latency\_percentiles": {  "values": {  "50.0": 87.5,  "95.0": 110.25,  "99.0": 111.65  }  }  }  ]  }  }  } |

### 实战-percentiles rank 以及网站访问时延 SLA 统计

课程大纲

SLA：就是你提供的服务的标准

我们的网站的提供的访问延时的SLA，确保所有的请求100%，都必须在200ms以内，大公司内，**一般都是要求100%在200ms以内**

如果超过1s，则需要升级到A级故障，代表网站的访问性能和用户体验急剧下降

需求：在200ms以内的，有百分之多少，在1000毫秒以内的有百分之多少，percentile ranks metric

这个percentile ranks，其实比pencentile还要常用

按照品牌分组，计算，电视机，售价在1000占比，2000占比，3000占比

GET /website/logs/\_search

{

"size": 0,

"aggs": {

"group\_by\_province": {

"terms": {

"field": "province"

},

"aggs": {

"latency\_percentile\_ranks": {

"percentile\_ranks": {

"field": "latency",

"values": [

200,

1000

]

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 38,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 12,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_province": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "新疆",  "doc\_count": 6,  "latency\_percentile\_ranks": {  "values": {  "200.0": 29.40613026819923,  "1000.0": 100  }  }  },  {  "key": "江苏",  "doc\_count": 6,  "latency\_percentile\_ranks": {  "values": {  "200.0": 100,  "1000.0": 100  }  }  }  ]  }  }  } |

percentile的优化

TDigest算法，用很多节点来执行百分比的计算，近似估计，有误差，节点越多，越精准

compression

限制节点数量最多 compression \* 20 = 2000个node去计算

默认100

越大，占用内存越多，越精准，性能越差

一个节点占用32字节，100 \* 20 \* 32 = 64KB

如果你想要percentile算法越精准，compression可以设置的越大

### 原理-doc value 正排索引的聚合内部原理

聚合分析的内部原理是什么？？？？aggs，term，metric avg max，执行一个聚合操作的时候，内部原理是怎样的呢？用了什么样的数据结构去执行聚合？是不是用的倒排索引？

我们对搜索+聚合来写个示例

GET /test\_index/test\_type/\_search

{

"query": {

"match": {

"search\_field": "test"

}

},

"aggs": {

"group\_by\_agg\_field": {

"terms": {

"field": "agg\_field"

}

}

}

}

假如说如果：纯用倒排索引来实现的弊端

反过来说：es肯定不是纯用倒排索引来实现聚合+搜索的

search\_field

有如下的 doc 数据：

doc1: hello world test1, test2

doc2: hello test

doc3: world test

建立倒排索引后：

hello doc1,doc2

world doc1,doc3

test1 doc1

test2 doc1

test doc2,doc3

如查询 test

"query": {

"match": {

"search\_field": "test"

}

}

test --> doc2,doc3 --> search result, doc2,doc3

那么对于聚合操作了？如agg\_field

doc2: agg1

doc3: agg2

100万个值

...

...

...

...

agg1 doc2

agg2 doc3

doc2, doc3, search result --> 实际上，要搜索到doc2的agg\_field的值是多少，doc3的agg\_field的值是多少

doc2和doc3的agg\_field的值之后，就可以根据值进行分组，实现terms bucket操作

doc2的agg\_field的值是多少，这个时候，如果你手上只有一个倒排索引，你该怎么办？？？你要扫描整个倒排索引，去一个一个的搜，拿到每个值，比如说agg1，看一下，它是不是doc2的值，拿到agg2,看一下，是不是doc2的值，直到找到doc2的agg\_field的值，在倒排索引中

如果用纯倒排索引去实现聚合，现实不现实啊？？？性能是很低下的。。。搜索，search，搜倒排索引，搜那个term，就结束了。。。聚合，搜索出了1万个doc，每个doc都要在倒排索引中搜索出它的那个聚合field的值

倒排索引+正排索引（doc value）的原理和优势

**doc value：正排索引，专门用于聚合运算的**

search\_field

doc1: hello world test1, test2

doc2: hello test

doc3: world test

hello doc1,doc2

world doc1,doc3

test1 doc1

test2 doc1

test doc2,doc3

"query": {

"match": {

"search\_field": "test"

}

}

test --> doc2,doc3 --> search result, doc2,doc3

如果是用倒排索引的话，必须遍历完整个倒排索引才可以。。。。

解释：因为可能你要聚合的那个field的值，是分词的，比如说hello world my name --> 一个doc的聚合field的值可能在倒排索引中对应多个value

所以说，当你在倒排索引中找到一个值，发现它是属于某个doc的时候，还不能停，必须遍历完整个倒排索引，才能说确保找到了每个doc对应的所有terms，然后进行分组聚合

doc value数据结构，正排索引：

...

...

...

100万个

doc2: agg1 hello world

doc3: agg2 test hello

我们有没有必要搜索完整个正排索引啊？？1万个doc --> 搜 -> 可能跟搜索到15000次，就搜索完了，就找到了1万个doc的聚合field的所有值了，然后就可以执行分组聚合操作了

### 原理-doc value机制内核级原理深入揭秘

1、doc value原理

（1）index-time生成

PUT/POST的时候，就会生成doc value数据，也就是正排索引

（2）核心原理与倒排索引类似

正排索引，也会写入磁盘文件中，然后呢，os cache先进行缓存，以提升访问doc value正排索引的性能

如果os cache内存大小不足够放得下整个正排索引，doc value，就会将doc value的数据写入磁盘文件中

（3）性能问题：给jvm更少内存，64g服务器，给jvm最多16g

es官方是建议，es大量是基于os cache来进行缓存和提升性能的，不建议用jvm内存来进行缓存，那样会导致一定的gc开销和oom问题

给jvm更少的内存，给os cache更大的内存

64g服务器，给jvm最多16g，几十个g的内存给os cache

os cache可以提升doc value和倒排索引的缓存和查询效率

2、column压缩

doc1: 550

doc2: 550

doc3: 500

合并相同值，550，doc1和doc2都保留一个550的标识即可

（1）所有值相同，直接保留单值

（2）少于256个值，使用table encoding模式：一种压缩方式

（3）大于256个值，看有没有最大公约数，有就除以最大公约数，然后保留这个最大公约数

doc1: 36

doc2: 24

6 --> doc1: 6, doc2: 4 --> 保留一个最大公约数6的标识，6也保存起来

（4）如果没有最大公约数，采取offset结合压缩的方式：

3、disable doc value

如果的确不需要doc value，比如聚合等操作，那么可以禁用，减少磁盘空间占用

PUT my\_index

{

"mappings": {

"my\_type": {

"properties": {

"my\_field": {

"type": "keyword"

"doc\_values": false

}

}

}

}

}

### 实战-string field 聚合实验以及 fielddata 原理初探

1、对于分词的field执行aggregation，发现报错。。。

GET /test\_index/test\_type/\_search

{

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": "test\_field"

}

}

}

}

|  |
| --- |
| {  "error": {  "root\_cause": [  {  "type": "illegal\_argument\_exception",  "reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."  }  ],  "type": "search\_phase\_execution\_exception",  "reason": "all shards failed",  "phase": "query",  "grouped": true,  "failed\_shards": [  {  "shard": 0,  "index": "test\_index",  "node": "4onsTYVZTjGvIj9\_spWz2w",  "reason": {  "type": "illegal\_argument\_exception",  "reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."  }  }  ],  "caused\_by": {  "type": "illegal\_argument\_exception",  "reason": "Fielddata is disabled on text fields by default. Set fielddata=true on [test\_field] in order to load fielddata in memory by uninverting the inverted index. Note that this can however use significant memory."  }  },  "status": 400  } |

对分词的field，直接执行聚合操作，会报错，大概意思是说，你必须要打开fielddata，然后将正排索引数据加载到内存中，才可以对分词的field执行聚合操作，而且会消耗很大的内存

2、给分词的field，设置fielddata=true，发现可以执行

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "text",

"fielddata": true

}

}

}

查看 mapping 关系如下：

{

"test\_index": {

"mappings": {

"test\_type": {

"properties": {

"test\_field": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

},

"fielddata": true

}

}

}

}

}

}

再次执行聚合操作，这个时候没有错误了。

GET /test\_index/test\_type/\_search

{

"size": 0,

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": "test\_field"

}

}

}

}

{

"took": 23,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 2,

"max\_score": 0,

"hits": []

},

"aggregations": {

"group\_by\_test\_field": {

"doc\_count\_error\_upper\_bound": 0,

"sum\_other\_doc\_count": 0,

"buckets": [

{

"key": "test",

"doc\_count": 2

}

]

}

}

}

如果要对分词的field执行聚合操作，必须将fielddata设置为true

3、使用内置field不分词，对string field进行聚合

GET /test\_index/test\_type/\_search

{

"size": 0,

"aggs": {

"group\_by\_test\_field": {

"terms": {

"field": **"test\_field.keyword"**

}

}

}

}

结果：

|  |
| --- |
| {  "took": 3,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_test\_field": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "test",  "doc\_count": 2  }  ]  }  }  } |

如果对不分词的field执行聚合操作，直接就可以执行，不需要设置fieldata=true

4、分词field+fielddata的工作原理

doc value --> 不分词的所有field，可以执行聚合操作 --> 如果你的某个field不分词，那么在index-time，就会自动生成doc value --> 针对这些不分词的field执行聚合操作的时候，自动就会用doc value来执行

分词field，是没有doc value的。。。在index-time，如果某个field是分词的，那么是不会给它建立doc value正排索引的，因为分词后，占用的空间过于大，所以默认是不支持分词field进行聚合的

分词field默认没有doc value，所以直接对分词field执行聚合操作，是会报错的

对于分词field，必须打开和使用fielddata，完全存在于纯内存中。。。结构和doc value类似。。。如果是ngram或者是大量term，那么必将占用大量的内存。。。

如果一定要对分词的field执行聚合，那么必须将fielddata=true，然后es就会在执行聚合操作的时候，现场将field对应的数据，建立一份fielddata正排索引，fielddata正排索引的结构跟doc value是类似的，但是只会讲fielddata正排索引加载到内存中来，然后基于内存中的fielddata正排索引执行分词field的聚合操作

如果直接对分词field执行聚合，报错，才会让我们开启fielddata=true，告诉我们，会将fielddata uninverted index，正排索引，加载到内存，会耗费内存空间

为什么fielddata必须在内存？因为大家自己思考一下，分词的字符串，需要按照term进行聚合，需要执行更加复杂的算法和操作，如果基于磁盘和os cache，那么性能会很差

### 原理-fielddata 内存控制一级 circuit breaker 短路器

1、fielddata核心原理

fielddata加载到内存的过程是lazy加载的，对一个analzyed field执行聚合时，才会加载，而且是field-level加载的

一个index的一个field，所有doc都会被加载，而不是少数doc

不是index-time创建，是query-time创建

2、fielddata内存限制

indices.fielddata.cache.size: 20%，超出限制，清除内存已有fielddata数据

fielddata占用的内存超出了这个比例的限制，那么就清除掉内存中已有的fielddata数据

默认无限制，限制内存使用，但是会导致频繁evict和reload，大量IO性能损耗，以及内存碎片和gc

3、监控fielddata内存使用

GET /\_stats/fielddata?fields=\*

GET /\_nodes/stats/indices/fielddata?fields=\*

GET /\_nodes/stats/indices/fielddata?level=indices&fields=\*

4、circuit breaker

如果一次query load的feilddata超过总内存，就会oom --> 内存溢出

circuit breaker会估算query要加载的fielddata大小，如果超出总内存，就短路，query直接失败

indices.breaker.fielddata.limit：fielddata的内存限制，默认60%

indices.breaker.request.limit：执行聚合的内存限制，默认40%

indices.breaker.total.limit：综合上面两个，限制在70%以内

### 配置-fielddata filter 的细粒度内存加载控制

POST /test\_index/\_mapping/my\_type

{

"properties": {

"my\_field": {

"type": "text",

"fielddata": {

"filter": {

"frequency": {

"min": 0.01,

"min\_segment\_size": 500

}

}

}

}

}

}

min：仅仅加载至少在1%的doc中出现过的term对应的fielddata

比如说某个值，hello，总共有1000个doc，hello必须在10个doc中出现，那么这个hello对应的fielddata才会加载到内存中来

min\_segment\_size：少于500 doc的segment不加载fielddata

加载fielddata的时候，也是按照segment去进行加载的，某个segment里面的doc数量少于500个，那么这个segment的fielddata就不加载

这个，就我的经验来看，有点底层了，一般不会去设置它，大家知道就好

### 原理-fielddata 预加载机制以及序号标记预加载

如果真的要对分词的field执行聚合，那么每次都在query-time现场生产fielddata并加载到内存中来，速度可能会比较慢

我们是不是可以预先生成加载fielddata到内存中来？？？

1、fielddata预加载

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "string",

"fielddata": {

"loading" : "eager"

}

}

}

}

query-time的fielddata生成和加载到内存，变为index-time，建立倒排索引的时候，会同步生成fielddata并且加载到内存中来，这样的话，对分词field的聚合性能当然会大幅度增强

2、序号标记预加载

global ordinal原理解释

doc1: status1

doc2: status2

doc3: status2

doc4: status1

有很多重复值的情况，会进行global ordinal标记

status1 --> 0

status2 --> 1

doc1: 0

doc2: 1

doc3: 1

doc4: 0

建立的fielddata也会是这个样子的，这样的好处就是减少重复字符串的出现的次数，减少内存的消耗

POST /test\_index/\_mapping/test\_type

{

"properties": {

"test\_field": {

"type": "string",

"fielddata": {

"loading" : "eager\_global\_ordinals"

}

}

}

}

### 优化-海量 bucket 优化机制：从深度优先到广度优先

当buckets数量特别多的时候，深度优先和广度优先的原理，图解

我们的数据，是每个演员的每个电影的评论

每个演员的评论的数量 --> 每个演员的每个电影的评论的数量

评论数量排名前10个的演员 --> 每个演员的电影取到评论数量排名前5的电影

{

"aggs" : {

"actors" : {

"terms" : {

"field" : "actors",

"size" : 10,

"collect\_mode" : "breadth\_first"

},

"aggs" : {

"costars" : {

"terms" : {

"field" : "films",

"size" : 5

}

}

}

}

}

}

深度优先的方式去执行聚合操作的

actor1 actor2 .... actor

film1 film2 film3 film1 film2 film3 ...film

比如说，我们有10万个actor，最后其实是主要10个actor就可以了

但是我们已经深度优先的方式，构建了一整颗完整的树出来了，10万个actor，每个actor平均有10部电影，10万 + 100万 --> 110万的数据量的一颗树

裁剪掉10万个actor中的99990 actor，99990 \* 10 = film，剩下10个actor，每个actor的10个film裁剪掉5个，110万 --> 10 \* 5 = 50个

构建了大量的数据，然后裁剪掉了99.99%的数据，浪费了

广度优先的方式去执行聚合

actor1 actor2 actor3 ..... n个actor

10万个actor，不去构建它下面的film数据，10万 --> 99990，10个actor，构建出film，裁剪出其中的5个film即可，10万 -> 50个

10倍

## 数据建模实战

就是生活中复杂的数据结构在 ES 中怎么建立

这里也是很多的例子

### 关系型与 document 类型数据模型对比

关系型数据库的数据模型 VS es的document数据模型

public class Department {

private Integer deptId;

private String name;

private String desc;

private List<Employee> employees;

}

public class Employee {

private Integer empId;

private String name;

private Integer age;

private String gender;

private Department dept; // 属于哪个部门

}

在关系型数据库中，如下两个表：

department表

dept\_id

name

desc

employee表

emp\_id

name

age

gender

dept\_id ## 部门 ID

三范式 --> 将每个数据实体拆分为一个独立的数据表，同时使用主外键关联关系将多个数据表关联起来 --> 确保没有任何冗余的数据

一份数据，只会放在一个数据表中 --> dept name，部门名称，就只会放在department表中，不会在employee表中也放一个dept name，如果说你要查看某个员工的部门名称，那么必须通过员工表中的外键，dept\_id，找到在部门表中对应的记录，然后找到部门名称

es文档数据模型

{

"deptId": "1",

"name": "研发部门",

"desc": "负责公司的所有研发项目",

"employees": [

{

"empId": "1",

"name": "张三",

"age": 28,

"gender": "男"

},

{

"empId": "2",

"name": "王兰",

"age": 25,

"gender": "女"

},

{

"empId": "3",

"name": "李四",

"age": 34,

"gender": "男"

}

]

}

es，更加类似于面向对象的数据模型，将所有由关联关系的数据，放在一个doc json类型数据中，整个数据的关系，还有完整的数据，都放在了一起

### 通过应用层 join 实现用户与博客的关联

1、构造用户与博客数据

在构造数据模型的时候，还是将有关联关系的数据，然后分割为不同的实体，类似于关系型数据库中的模型

案例背景：博客网站， 我们会模拟各种用户发表各种博客，然后针对用户和博客之间的关系进行数据建模，同时针对建模好的数据执行各种搜索/聚合的操作

PUT /website/users/1

{

"name": "小鱼儿",

"email": "xiaoyuer@sina.com",

"birthday": "1980-01-01"

}

PUT /website/blogs/1

{

"title": "我的第一篇博客",

"content": "这是我的第一篇博客，开通啦！！！"

**"userId": 1**

}

一个用户对应多个博客，一对多的关系，做了建模

建模方式，分割实体，类似三范式的方式，用主外键关联关系，将多个实体关联起来

2、搜索小鱼儿发表的所有博客

GET /website/users/\_search

{

"query": {

"term": {

"name.keyword": {

"value": "小鱼儿"

}

}

}

}

|  |
| --- |
| {  "took": 91,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 0.2876821,  "hits": [  {  "\_index": "website",  "\_type": "users",  "\_id": "1",  "\_score": 0.2876821,  "\_source": {  "name": "小鱼儿",  "email": "xiaoyuer@sina.com",  "birthday": "1980-01-01"  }  }  ]  }  } |

比如这里搜索的是，1万个用户的博客，可能第一次搜索，会得到1万个userId

GET /website/blogs/\_search

{

"query": {

"constant\_score": {

"filter": {

"terms": {

"userId": [

1

]

}

}

}

}

}

第二次搜索的时候，要放入terms中1万个userId，才能进行搜索，这个时候性能比较差了

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "website",

"\_type": "blogs",

"\_id": "1",

"\_score": 1,

"\_source": {

"title": "小鱼儿的第一篇博客",

"content": "大家好，我是小鱼儿，这是我写的第一篇博客！",

"userId": 1

}

}

]

}

}

上面的操作，就属于应用层的join，在应用层先查出一份数据，然后再查出一份数据，进行关联

3、优点和缺点

优点：数据不冗余，维护方便

缺点：应用层join，如果关联数据过多，导致查询过大，性能很差

### 通过数据冗余实现用户与博客的关联

1、构造冗余的用户和博客数据

第二种建模方式：用冗余数据，采用文档数据模型，进行数据建模，实现用户和博客的关联

PUT /website/users/1

{

"name": "小鱼儿",

"email": "xiaoyuer@sina.com",

"birthday": "1980-01-01"

}

PUT /website/blogs/1

{

"title": "小鱼儿的第一篇博客",

"content": "大家好，我是小鱼儿。。。",

"userInfo": {

"userId": 1,

"username": "小鱼儿"

}

}

冗余数据，就是说，将可能会进行搜索的条件和要搜索的数据，放在一个doc中

2、基于冗余用户数据搜索博客

GET /website/blogs/\_search

{

"query": {

"term": {

"userInfo.username.keyword": {

"value": "小鱼儿"

}

}

}

}

就不需要走应用层的join，先搜一个数据，找到id，再去搜另一份数据

直接走一个有冗余数据的type即可，指定要的搜索条件，即可搜索出自己想要的数据来

3、优点和缺点

优点：性能高，不需要执行两次搜索

缺点：数据冗余，维护成本高 --> 每次如果你的username变化了，同时要更新user type和blog type

一般来说，对于es这种NoSQL类型的数据存储来讲，都是冗余模式....

当然，你要去维护数据的关联关系，也是很有必要的，所以一旦出现冗余数据的修改，必须记得将所有关联的数据全部更新

### 对每个用户发表的博客进行分组

1、构造更多测试数据

PUT /website/users/3

{

"name": "黄药师",

"email": "huangyaoshi@sina.com",

"birthday": "1970-10-24"

}

PUT /website/blogs/3

{

"title": "我是黄药师",

"content": "我是黄药师啊，各位同学们！！！",

"userInfo": {

"userId": 1,

"userName": "黄药师"

}

}

PUT /website/users/2

{

"name": "花无缺",

"email": "huawuque@sina.com",

"birthday": "1980-02-02"

}

PUT /website/blogs/4

{

"title": "花无缺的身世揭秘",

"content": "大家好，我是花无缺，所以我的身世是。。。",

"userInfo": {

"userId": 2,

"userName": "花无缺"

}

}

2、对每个用户发表的博客进行分组

比如说，小鱼儿发表的那些博客，花无缺发表了哪些博客，黄药师发表了哪些博客

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"group\_by\_username": {

"terms": {

"field": "userInfo.username.keyword"

},

"aggs": {

"top\_blogs": {

"top\_hits": {

"\_source": {

"include": "title"

},

"size": 5

}

}

}

}

}

}

### 对文件系统系统进行数据建模以及文件搜索实战

数据建模，对类似文件系统这种的有多层级关系的数据进行建模

参考资料：

https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-pathhierarchy-tokenizer.html

1、文件系统数据构造

PUT /fs

{

"settings": {

"analysis": {

"analyzer": {

"paths": {

"tokenizer": **"path\_hierarchy"**

}

}

}

}

}

path\_hierarchy tokenizer讲解：

/a/b/c/d --> path\_hierarchy -> /a/b/c/d, /a/b/c, /a/b, /a

fs: filesystem

PUT /fs/\_mapping/file

{

"properties": {

"name": {

"type": "keyword"

},

"path": {

"type": "keyword",

"fields": {

"tree": {

"type": "text",

"analyzer": "paths"

}

}

}

}

}

PUT /fs/file/1

{

"name": "README.txt",

"path": "/workspace/projects/helloworld",

"contents": "这是我的第一个elasticsearch程序"

}

2、对文件系统执行搜索

文件搜索需求1：查找一份，内容包括elasticsearch，在/workspace/projects/hellworld这个目录下的文件

GET /fs/file/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"contents": "elasticsearch"

}

},

{

"constant\_score": {

"filter": {

"term": {

"path": "/workspace/projects/helloworld"

}

}

}

}

]

}

}

}

结果：

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1.284885,  "hits": [  {  "\_index": "fs",  "\_type": "file",  "\_id": "1",  "\_score": 1.284885,  "\_source": {  "name": "README.txt",  "path": "/workspace/projects/helloworld",  "contents": "这是我的第一个elasticsearch程序"  }  }  ]  }  } |

搜索需求2：搜索/workspace目录下，内容包含elasticsearch的所有的文件

/workspace/projects/helloworld doc1

/workspace/projects doc1

/workspace doc1

GET /fs/file/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"contents": "elasticsearch"

}

},

{

"constant\_score": {

"filter": {

"term": {

**"path.tree": "/workspace"**

}

}

}

}

]

}

}

}

### 锁-基于全局锁实现悲观锁并发控制

1、悲观锁的简要说明

基于version的乐观锁并发控制

在数据建模，结合文件系统建模的这个案例，把悲观锁的并发控制，3种锁粒度，都给大家仔细讲解一下

最粗的一个粒度，全局锁

/workspace/projects/helloworld

如果多个线程，都过来，要并发地给/workspace/projects/helloworld下的README.txt修改文件名

实际上要进行并发的控制，避免出现多线程的并发安全问题，比如多个线程修改，纯并发，先执行的修改操作被后执行的修改操作给覆盖了

get current version

带着这个current version去执行修改，如果一旦发现数据已经被别人给修改了，version号跟之前自己获取的已经不一样了; 那么必须重新获取新的version号再次尝试修改

上来就尝试给这条数据加个锁，然后呢，此时就只有你能执行各种各样的操作了，其他人不能执行操作

第一种锁：全局锁，直接锁掉整个fs index

2、全局锁的上锁实验

如执行下面的命令：

PUT /fs/lock/global/\_create

{}

可以看见是执行成功的。

fs: 你要上锁的那个index

lock: 就是你指定的一个对这个index上全局锁的一个type

global: 就是你上的全局锁对应的这个doc的id

\_create：强制必须是创建，如果/fs/lock/global这个doc已经存在，那么创建失败，报错

利用了doc来进行上锁

/fs/lock/global /index/type/id --> doc

{

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

另外一个线程同时尝试上锁

PUT /fs/lock/global/\_create

{}

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][global]: version conflict, document already exists (current version [1])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "2",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][global]: version conflict, document already exists (current version [1])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "2",

"index": "fs"

},

"status": 409

}

如果失败，就再次重复尝试上锁

执行各种操作。。。

POST /fs/file/1/\_update

{

"doc": {

"name": "README1.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 2,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

DELETE /fs/lock/global

{

"found": true,

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 2,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

另外一个线程，因为之前发现上锁失败，反复尝试重新上锁，终于上锁成功了，因为之前获取到全局锁的那个线程已经delete /fs/lock/global全局锁了

PUT /fs/lock/global/\_create

{}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "global",

"\_version": 3,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"created": true

}

POST /fs/file/1/\_update

{

"doc": {

"name": "README.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 3,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

DELETE /fs/lock/global

3、全局锁的优点和缺点

优点：操作非常简单，非常容易使用，成本低

缺点：你直接就把整个index给上锁了，这个时候对index中所有的doc的操作，都会被block住，导致整个系统的并发能力很低

上锁解锁的操作不是频繁，然后每次上锁之后，执行的操作的耗时不会太长，用这种方式，方便

### 锁-基于 document 锁实现悲观锁并发控制

1、对document level锁，详细的讲解

全局锁，一次性就锁整个index，对这个index的所有增删改操作都会被block住，如果上锁不频繁，还可以，比较简单

细粒度的一个锁，document锁，顾名思义，每次就锁你要操作的，你要执行增删改的那些doc，doc锁了，其他线程就不能对这些doc执行增删改操作了

但是你只是锁了部分doc，其他线程对其他的doc还是可以上锁和执行增删改操作的

document锁，是用脚本进行上锁

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 123

}

}

/fs/lock，是固定的，就是说fs下的lock type，专门用于进行上锁

/fs/lock/id，比如1，id其实就是你要上锁的那个doc的id，代表了某个doc数据对应的lock（也是一个doc）

\_update + upsert：执行upsert操作

params，里面有个process\_id，process\_id，是你的要执行增删改操作的进程的唯一id，比如说可以在java系统，启动的时候，给你的每个线程都用UUID自动生成一个thread id，你的系统进程启动的时候给整个进程也分配一个UUID。process\_id + thread\_id就代表了某一个进程下的某个线程的唯一标识。可以自己用UUID生成一个唯一ID

process\_id很重要，会在lock中，设置对对应的doc加锁的进程的id，这样其他进程过来的时候，才知道，这条数据已经被别人给锁了

assert false，不是当前进程加锁的话，则抛出异常

ctx.op='noop'，不做任何修改

如果该document之前没有被锁，/fs/lock/1之前不存在，也就是doc id=1没有被别人上过锁; upsert的语法，那么执行index操作，创建一个/fs/lock/id这条数据，而且用params中的数据作为这个lock的数据。process\_id被设置为123，script不执行。这个时候象征着process\_id=123的进程已经锁了一个doc了。

如果document被锁了，就是说/fs/lock/1已经存在了，代表doc id=1已经被某个进程给锁了。那么执行update操作，script，此时会比对process\_id，如果相同，就是说，某个进程，之前锁了这个doc，然后这次又过来，就可以直接对这个doc执行操作，说明是该进程之前锁的doc，则不报错，不执行任何操作，返回success; 如果process\_id比对不上，说明doc被其他doc给锁了，此时报错

/fs/lock/1

{

"process\_id": 123

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 123

}

}

script：ctx.\_source.process\_id，123

process\_id：加锁的upsert请求中带过来额proess\_id

如果两个process\_id相同，说明是一个进程先加锁，然后又过来尝试加锁，可能是要执行另外一个操作，此时就不会block，对同一个process\_id是不会block，ctx.op= 'noop'，什么都不做，返回一个success

如果说已经有一个进程加了锁了

/fs/lock/1

{

"process\_id": 123

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

"params": {

"process\_id": 234

}

}

"script": "if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';"

ctx.\_source.process\_id：123

process\_id: 234

process\_id不相等，说明这个doc之前已经被别人上锁了，process\_id=123上锁了; process\_id=234过来再次尝试上锁，失败，assert false，就会报错

此时遇到报错的process，就应该尝试重新上锁，直到上锁成功

有报错的话，如果有些doc被锁了，那么需要重试

直到所有锁定都成功，执行自己的操作。。。

释放所有的锁

2、上document锁的完整实验过程

scripts/judge-lock.groovy: if ( ctx.\_source.process\_id != process\_id ) { assert false }; ctx.op = 'noop';

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 123

}

}

}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"result": "created",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

GET /fs/lock/1

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"found": true,

"\_source": {

"process\_id": 123

}

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 234 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 234

}

}

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:data/write/update[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "failed to execute script",

"caused\_by": {

"type": "script\_exception",

"reason": "error evaluating judge-lock",

"caused\_by": {

"type": "power\_assertion\_error",

"reason": "assert false\n"

},

"script\_stack": [],

"script": "",

"lang": "groovy"

}

},

"status": 400

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 123 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 123

}

}

}

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 1,

"result": "noop",

"\_shards": {

"total": 0,

"successful": 0,

"failed": 0

}

}

POST /fs/file/1/\_update

{

"doc": {

"name": "README1.txt"

}

}

{

"\_index": "fs",

"\_type": "file",

"\_id": "1",

"\_version": 4,

"result": "updated",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

}

}

POST /fs/\_refresh

GET /fs/lock/\_search?scroll=1m

{

"query": {

"term": {

"process\_id": 123

}

}

}

{

"\_scroll\_id": "DnF1ZXJ5VGhlbkZldGNoBQAAAAAAACPkFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAj5RY0b25zVFlWWlRqR3ZJajlfc3BXejJ3AAAAAAAAI-YWNG9uc1RZVlpUakd2SWo5X3NwV3oydwAAAAAAACPnFjRvbnNUWVZaVGpHdklqOV9zcFd6MncAAAAAAAAj6BY0b25zVFlWWlRqR3ZJajlfc3BXejJ3",

"took": 51,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_score": 1,

"\_source": {

"process\_id": 123

}

}

]

}

}

PUT /fs/lock/\_bulk

{ "delete": { "\_id": 1}}

{

"took": 20,

"errors": false,

"items": [

{

"delete": {

"found": true,

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 2,

"result": "deleted",

"\_shards": {

"total": 2,

"successful": 1,

"failed": 0

},

"status": 200

}

}

]

}

POST /fs/lock/1/\_update

{

"upsert": { "process\_id": 234 },

"script": {

"lang": "groovy",

"file": "judge-lock",

"params": {

"process\_id": 234

}

}

}

process\_id=234上锁就成功了

### 锁-基于共享锁和排它锁实现悲观锁并发控制

1、共享锁和排他锁的说明

共享锁：这份数据是共享的，然后多个线程过来，都可以获取同一个数据的共享锁，然后对这个数据执行读操作

排他锁：是排他的操作，只能一个线程获取排他锁，然后执行增删改操作

读写锁的分离

如果只是要读取数据的话，那么任意个线程都可以同时进来然后读取数据，每个线程都可以上一个共享锁

但是这个时候，如果有线程要过来修改数据，那么会尝试上排他锁，排他锁会跟共享锁互斥，也就是说，如果有人已经上了共享锁了，那么排他锁就不能上，就得等

如果有人在读数据，就不允许别人来修改数据

反之，也是一样的

如果有人在修改数据，就是加了排他锁

那么其他线程过来要修改数据，也会尝试加排他锁，此时会失败，锁冲突，必须等待，同时只能有一个线程修改数据

如果有人过来同时要读取数据，那么会尝试加共享锁，此时会失败，因为共享锁和排他锁是冲突的

如果有在修改数据，就不允许别人来修改数据，也不允许别人来读取数据

2、共享锁和排他锁的实验

第一步：有人在读数据，其他人也能过来读数据

judge-lock-2.groovy: if (ctx.\_source.lock\_type == 'exclusive') { assert false }; ctx.\_source.lock\_count++

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

GET /fs/lock/1

{

"\_index": "fs",

"\_type": "lock",

"\_id": "1",

"\_version": 3,

"found": true,

"\_source": {

"lock\_type": "shared",

"lock\_count": 3

}

}

就给大家模拟了，有人上了共享锁，你还是要上共享锁，直接上就行了，没问题，只是lock\_count加1

2、已经有人上了共享锁，然后有人要上排他锁

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

排他锁用的不是upsert语法，create语法，要求lock必须不能存在，直接自己是第一个上锁的人，上的是排他锁

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [3])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [3])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

},

"status": 409

}

如果已经有人上了共享锁，明显/fs/lock/1是存在的，create语法去上排他锁，肯定会报错

3、对共享锁进行解锁

POST /fs/lock/1/\_update

{

"script": {

"lang": "groovy",

"file": "unlock-shared"

}

}

连续解锁3次，此时共享锁就彻底没了

每次解锁一个共享锁，就对lock\_count先减1，如果减了1之后，是0，那么说明所有的共享锁都解锁完了，此时就就将/fs/lock/1删除，就彻底解锁所有的共享锁

3、上排他锁，再上排他锁

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

其他线程

PUT /fs/lock/1/\_create

{ "lock\_type": "exclusive" }

{

"error": {

"root\_cause": [

{

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [7])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

}

],

"type": "version\_conflict\_engine\_exception",

"reason": "[lock][1]: version conflict, document already exists (current version [7])",

"index\_uuid": "IYbj0OLGQHmMUpLfbhD4Hw",

"shard": "3",

"index": "fs"

},

"status": 409

}

4、上排他锁，上共享锁

POST /fs/lock/1/\_update

{

"upsert": {

"lock\_type": "shared",

"lock\_count": 1

},

"script": {

"lang": "groovy",

"file": "judge-lock-2"

}

}

{

"error": {

"root\_cause": [

{

"type": "remote\_transport\_exception",

"reason": "[4onsTYV][127.0.0.1:9300][indices:data/write/update[s]]"

}

],

"type": "illegal\_argument\_exception",

"reason": "failed to execute script",

"caused\_by": {

"type": "script\_exception",

"reason": "error evaluating judge-lock-2",

"caused\_by": {

"type": "power\_assertion\_error",

"reason": "assert false\n"

},

"script\_stack": [],

"script": "",

"lang": "groovy"

}

},

"status": 400

}

5、解锁排他锁

DELETE /fs/lock/1

### 基于 nested object 实现博客与评论嵌套关系

1、做一个实验，引出来为什么需要nested object

冗余数据方式的来建模，其实用的就是object类型，我们这里又要引入一种新的object类型，nested object类型

博客，评论，做的这种数据模型

PUT /website/blogs/6

{

"title": "花无缺发表的一篇帖子",

"content": "我是花无缺，大家要不要考虑一下投资房产和买股票的事情啊。。。",

"tags": [ "投资", "理财" ],

"comments": [

{

"name": "小鱼儿",

"comment": "什么股票啊？推荐一下呗",

"age": 28,

"stars": 4,

"date": "2016-09-01"

},

{

"name": "黄药师",

"comment": "我喜欢投资房产，风，险大收益也大",

"age": 31,

"stars": 5,

"date": "2016-10-22"

}

]

}

搜索条件：年龄是28岁的黄药师评论过的博客，搜索

GET /website/blogs/\_search

{

"query": {

"bool": {

"must": [

{ "match": { "comments.name": "黄药师" }},

{ "match": { "comments.age": 28 }}

]

}

}

}

结果：

|  |
| --- |
| {  "took": 102,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1.8022683,  "hits": [  {  "\_index": "website",  "\_type": "blogs",  "\_id": "6",  "\_score": 1.8022683,  "\_source": {  "title": "花无缺发表的一篇帖子",  "content": "我是花无缺，大家要不要考虑一下投资房产和买股票的事情啊。。。",  "tags": [  "投资",  "理财"  ],  "comments": [  {  "name": "小鱼儿",  "comment": "什么股票啊？推荐一下呗",  "age": 28,  "stars": 4,  "date": "2016-09-01"  },  {  "name": "黄药师",  "comment": "我喜欢投资房产，风，险大收益也大",  "age": 31,  "stars": 5,  "date": "2016-10-22"  }  ]  }  }  ]  }  } |

结果是。。。好像不太对啊？？？

object类型数据结构的底层存储。。。，如上面的数据的存储结构是如下类似的：

{

"title": [ "花无缺", "发表", "一篇", "帖子" ],

"content": [ "我", "是", "花无缺", "大家", "要不要", "考虑", "一下", "投资", "房产", "买", "股票", "事情" ],

"tags": [ "投资", "理财" ],

"comments.name": [ "小鱼儿", "黄药师" ],

"comments.comment": [ "什么", "股票", "推荐", "我", "喜欢", "投资", "房产", "风险", "收益", "大" ],

"comments.age": [ 28, 31 ],

"comments.stars": [ 4, 5 ],

"comments.date": [ 2016-09-01, 2016-10-22 ]

}

object类型底层数据结构，会将一个json数组中的数据，进行扁平化

所以，直接命中了这个document，name=黄药师，age=28，正好符合

2、引入nested object类型，来解决object类型底层数据结构导致的问题

**修改mapping，将comments的类型从object设置为nested**

PUT /website

{

"mappings": {

"blogs": {

"properties": {

"comments": {

"type": "nested",

"properties": {

"name": { "type": "string" },

"comment": { "type": "string" },

"age": { "type": "short" },

"stars": { "type": "short" },

"date": { "type": "date" }

}

}

}

}

}

}

这样定义后就会被存成如下的数据格式：

{

"comments.name": [ "小鱼儿" ],

"comments.comment": [ "什么", "股票", "推荐" ],

"comments.age": [ 28 ],

"comments.stars": [ 4 ],

"comments.date": [ 2014-09-01 ]

}

{

"comments.name": [ "黄药师" ],

"comments.comment": [ "我", "喜欢", "投资", "房产", "风险", "收益", "大" ],

"comments.age": [ 31 ],

"comments.stars": [ 5 ],

"comments.date": [ 2014-10-22 ]

}

{

"title": [ "花无缺", "发表", "一篇", "帖子" ],

"body": [ "我", "是", "花无缺", "大家", "要不要", "考虑", "一下", "投资", "房产", "买", "股票", "事情" ],

"tags": [ "投资", "理财" ]

}

再次搜索，成功了。。。

GET /website/blogs/\_search

{

"query": {

"bool": {

"must": [

{

"match": {

"title": "花无缺"

}

},

{

"nested": {

"path": "comments",

"query": {

"bool": {

"must": [

{

"match": {

"comments.name": "黄药师"

}

},

{

"match": {

"comments.age": 28

}

}

]

}

}

}

}

]

}

}

}

score\_mode：max，min，avg，none，默认是avg

如果搜索命中了多个nested document，如何讲个多个nested document的分数合并为一个分数

### 对嵌套的博客评论数据进行聚合分析

我们讲解一下基于nested object中的数据进行聚合分析

聚合数据分析的需求1：按照评论日期进行bucket划分，然后拿到每个月的评论的评分的平均值

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"comments\_path": {

"nested": {

"path": "comments"

},

"aggs": {

"group\_by\_comments\_date": {

"date\_histogram": {

"field": "comments.date",

"interval": "month",

"format": "yyyy-MM"

},

"aggs": {

"avg\_stars": {

"avg": {

"field": "comments.stars"

}

}

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 52,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0,  "hits": []  },  "aggregations": {  "comments\_path": {  "doc\_count": 4,  "group\_by\_comments\_date": {  "buckets": [  {  "key\_as\_string": "2016-08",  "key": 1470009600000,  "doc\_count": 1,  "avg\_stars": {  "value": 3  }  },  {  "key\_as\_string": "2016-09",  "key": 1472688000000,  "doc\_count": 2,  "avg\_stars": {  "value": 4.5  }  },  {  "key\_as\_string": "2016-10",  "key": 1475280000000,  "doc\_count": 1,  "avg\_stars": {  "value": 5  }  }  ]  }  }  }  } |

按年龄聚合：

GET /website/blogs/\_search

{

"size": 0,

"aggs": {

"comments\_path": {

"nested": {

"path": "comments"

},

"aggs": {

"group\_by\_comments\_age": {

"histogram": {

"field": "comments.age",

"interval": 10

},

"aggs": {

"reverse\_path": {

"reverse\_nested": {},

"aggs": {

"group\_by\_tags": {

"terms": {

"field": "tags.keyword"

}

}

}

}

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 0,  "hits": []  },  "aggregations": {  "comments\_path": {  "doc\_count": 4,  "group\_by\_comments\_age": {  "buckets": [  {  "key": 20,  "doc\_count": 1,  "reverse\_path": {  "doc\_count": 1,  "group\_by\_tags": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "投资",  "doc\_count": 1  },  {  "key": "理财",  "doc\_count": 1  }  ]  }  }  },  {  "key": 30,  "doc\_count": 3,  "reverse\_path": {  "doc\_count": 2,  "group\_by\_tags": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "大侠",  "doc\_count": 1  },  {  "key": "投资",  "doc\_count": 1  },  {  "key": "理财",  "doc\_count": 1  },  {  "key": "练功",  "doc\_count": 1  }  ]  }  }  }  ]  }  }  }  } |

### parent-child-研发中心管理案例以及福字关系数据建模

nested object的建模，有个不好的地方，就是采取的是类似冗余数据的方式，将多个数据都放在一起了，维护成本就比较高

parent child建模方式，采取的是类似于关系型数据库的三范式类的建模，多个实体都分割开来，每个实体之间都通过一些关联方式，进行了父子关系的关联，各种数据不需要都放在一起，父doc和子doc分别在进行更新的时候，都不会影响对方

一对多关系的建模，维护起来比较方便，而且我们之前说过，类似关系型数据库的建模方式，应用层join的方式，会导致性能比较差，因为做多次搜索。父子关系的数据模型，不会，性能很好。因为虽然数据实体之间分割开来，但是我们在搜索的时候，由es自动为我们处理底层的关联关系，并且通过一些手段保证搜索性能。

父子关系数据模型，相对于nested数据模型来说，优点是父doc和子doc互相之间不会影响

要点：父子关系元数据映射，用于确保查询时候的高性能，但是有一个限制，就是父子数据必须存在于一个shard中

父子关系数据存在一个shard中，而且还有映射其关联关系的元数据，那么搜索父子关系数据的时候，不用跨分片，一个分片本地自己就搞定了，性能当然高咯

案例背景：研发中心员工管理案例，一个IT公司有多个研发中心，每个研发中心有多个员工

对 employee 数据添加\_parent属性：

PUT /company

{

"mappings": {

"rd\_center": {},

"employee": {

**"\_parent":** {

"type": "rd\_center"

}

}

}

}

父子关系建模的核心，多个type之间有父子关系，用\_parent指定父type

POST /company/rd\_center/\_bulk

{ "index": { "\_id": "1" }}

{ "name": "北京研发总部", "city": "北京", "country": "中国" }

{ "index": { "\_id": "2" }}

{ "name": "上海研发中心", "city": "上海", "country": "中国" }

{ "index": { "\_id": "3" }}

{ "name": "硅谷人工智能实验室", "city": "硅谷", "country": "美国" }

shard路由的时候，id=1的rd\_center doc，默认会根据id进行路由，到某一个shard

PUT /company/employee/1?**parent=1**

{

"name": "张三",

"birthday": "1970-10-24",

"hobby": "爬山"

}

维护父子关系的核心，parent=1，指定了这个数据的父doc的id

此时，parent-child关系，就确保了说，父doc和子doc都是保存在一个shard上的。内部原理还是doc routing，employee和rd\_center的数据，都会用parent id作为routing，这样就会到**一个shard**

就不会根据id=1的employee doc的id进行路由了，而是**根据parent=1进行路由**，会根据父doc的id进行路由，那么就可以通过底层的路由机制，保证父子数据存在于一个shard中

POST /company/employee/\_bulk

{ "index": { "\_id": 2, "parent": "1" }}

{ "name": "李四", "birthday": "1982-05-16", "hobby": "游泳" }

{ "index": { "\_id": 3, "parent": "2" }}

{ "name": "王二", "birthday": "1979-04-01", "hobby": "爬山" }

{ "index": { "\_id": 4, "parent": "3" }}

{ "name": "赵五", "birthday": "1987-05-11", "hobby": "骑马" }

### 根据员工信息和研发中心互相搜索父子数据

我们已经建立了父子关系的数据模型之后，就要基于这个模型进行各种搜索和聚合了

1、搜索有1980年以后出生的员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

**"has\_child"**: {

"type": "employee",

"query": {

"range": {

"birthday": {

"gte": "1980-01-01"

}

}

}

}

}

}

|  |
| --- |
| {  "took": 33,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 2,  "max\_score": 1,  "hits": [  {  "\_index": "company",  "\_type": "rd\_center",  "\_id": "1",  "\_score": 1,  "\_source": {  "name": "北京研发总部",  "city": "北京",  "country": "中国"  }  },  {  "\_index": "company",  "\_type": "rd\_center",  "\_id": "3",  "\_score": 1,  "\_source": {  "name": "硅谷人工智能实验室",  "city": "硅谷",  "country": "美国"  }  }  ]  }  } |

2、搜索有名叫张三的员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

"has\_child": {

"type": "employee",

"query": {

"match": {

"name": "张三"

}

}

}

}

}

|  |
| --- |
| {  "took": 2,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1,  "hits": [  {  "\_index": "company",  "\_type": "rd\_center",  "\_id": "1",  "\_score": 1,  "\_source": {  "name": "北京研发总部",  "city": "北京",  "country": "中国"  }  }  ]  }  } |

3、搜索有至少2个以上员工的研发中心

GET /company/rd\_center/\_search

{

"query": {

"has\_child": {

"type": "employee",

"min\_children": 2,

"query": {

"match\_all": {}

}

}

}

}

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1,  "hits": [  {  "\_index": "company",  "\_type": "rd\_center",  "\_id": "1",  "\_score": 1,  "\_source": {  "name": "北京研发总部",  "city": "北京",  "country": "中国"  }  }  ]  }  } |

4、搜索在中国的研发中心的员工

GET /company/employee/\_search

{

"query": {

"has\_parent": {

"parent\_type": "rd\_center",

"query": {

"term": {

"country.keyword": "中国"

}

}

}

}

}

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1,  "hits": [  {  "\_index": "company",  "\_type": "employee",  "\_id": "3",  "\_score": 1,  "\_routing": "2",  "\_parent": "2",  "\_source": {  "name": "王二",  "birthday": "1979-04-01",  "hobby": "爬山"  }  },  {  "\_index": "company",  "\_type": "employee",  "\_id": "1",  "\_score": 1,  "\_routing": "1",  "\_parent": "1",  "\_source": {  "name": "张三",  "birthday": "1970-10-24",  "hobby": "爬山"  }  },  {  "\_index": "company",  "\_type": "employee",  "\_id": "2",  "\_score": 1,  "\_routing": "1",  "\_parent": "1",  "\_source": {  "name": "李四",  "birthday": "1982-05-16",  "hobby": "游泳"  }  }  ]  }  } |

### 对每个国家的员工兴趣爱好进行聚合分析

统计每个国家的喜欢每种爱好的员工有多少个

GET /company/rd\_center/\_search

{

"size": 0,

"aggs": {

"group\_by\_country": {

"terms": {

"field": "country.keyword"

},

"aggs": {

"group\_by\_child\_employee": {

"children": {

"type": "employee"

},

"aggs": {

"group\_by\_hobby": {

"terms": {

"field": "hobby.keyword"

}

}

}

}

}

}

}

}

结果：

|  |
| --- |
| {  "took": 15,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 0,  "hits": []  },  "aggregations": {  "group\_by\_country": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "中国",  "doc\_count": 2,  "group\_by\_child\_employee": {  "doc\_count": 3,  "group\_by\_hobby": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "爬山",  "doc\_count": 2  },  {  "key": "游泳",  "doc\_count": 1  }  ]  }  }  },  {  "key": "美国",  "doc\_count": 1,  "group\_by\_child\_employee": {  "doc\_count": 1,  "group\_by\_hobby": {  "doc\_count\_error\_upper\_bound": 0,  "sum\_other\_doc\_count": 0,  "buckets": [  {  "key": "骑马",  "doc\_count": 1  }  ]  }  }  }  ]  }  }  } |

### 祖孙三代数据关系建模以及搜索实战

父子关系，祖孙三层关系的数据建模，搜索

PUT /company

{

"mappings": {

"country": {},

"rd\_center": {

"\_parent": {

"type": "country"

}

},

"employee": {

"\_parent": {

"type": "rd\_center"

}

}

}

}

country -> rd\_center -> employee，祖孙三层数据模型

POST /company/country/\_bulk

{ "index": { "\_id": "1" }}

{ "name": "中国" }

{ "index": { "\_id": "2" }}

{ "name": "美国" }

POST /company/rd\_center/\_bulk

{ "index": { "\_id": "1", "parent": "1" }}

{ "name": "北京研发总部" }

{ "index": { "\_id": "2", "parent": "1" }}

{ "name": "上海研发中心" }

{ "index": { "\_id": "3", "parent": "2" }}

{ "name": "硅谷人工智能实验室" }

PUT /company/employee/1?**parent=1**&**routing=1**

{

"name": "张三",

"dob": "1970-10-24",

"hobby": "爬山"

}

routing参数的讲解，必须跟grandparent相同，否则有问题

country，用的是自己的id去路由; rd\_center，parent，用的是country的id去路由; employee，如果也是仅仅指定一个parent，那么用的是rd\_center的id去路由，这就导致祖孙三层数据不会在一个shard上

孙子辈儿，要手动指定routing，指定为爷爷辈儿的数据的id

案例：搜索有爬山爱好的员工所在的国家

GET /company/country/\_search

{

"query": {

"has\_child": {

"type": "rd\_center",

"query": {

"has\_child": {

"type": "employee",

"query": {

"match": {

"hobby": "爬山"

}

}

}

}

}

}

}

|  |
| --- |
| {  "took": 10,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1,  "hits": [  {  "\_index": "company",  "\_type": "country",  "\_id": "1",  "\_score": 1,  "\_source": {  "name": "中国"  }  }  ]  }  } |

## Elasticsearch 高手进阶

### 基于term vectors深入探查数据的情况

1、term vector介绍

概括来说就是：获取document中的某个field内的各个term的统计信息

term information: term frequency in the field, term positions, start and end offsets, term payloads

term statistics: 设置term\_statistics=true;

total term frequency： 一个term在所有document中出现的频率;

document frequency：有多少document包含这个term

field statistics: document count，有多少document包含这个field;

sum of document frequency：一个field中所有term的df之和;

sum of total term frequency：一个field中的所有term的tf之和

怎么查看 termvector，用如下的命令：

GET /twitter/tweet/1/\_termvectors

GET /twitter/tweet/1/\_termvectors?fields=text

term statistics和field statistics并不精准，不会被考虑有的doc可能被删除了

我告诉大家，其实很少用，用的时候，一般来说，就是你需要对一些数据做探查的时候。比如说，你想要看到某个term，某个词条，大话西游，这个词条，在多少个document中出现了。或者说某个field，film\_desc，电影的说明信息，有多少个doc包含了这个说明信息。

2、index-time term vector实验

term vector，涉及了很多的term和field相关的统计信息，有两种方式可以采集到这个统计信息

（1）index-time：你在mapping里配置一下，然后建立索引的时候，就直接给你生成这些term和field的统计信息了

（2）query-time：你之前没有生成过任何的Term vector信息，然后在查看term vector的时候，直接就可以看到了，会on the fly，现场计算出各种统计信息，然后返回给你

这一讲，不会手敲任何命令，直接copy我做好的命令，因为这一讲的重点，不是掌握什么搜索或者聚合的语法，而是说，掌握，如何采集term vector信息，然后如何看懂term vector信息，你能掌握利用term vector进行数据探查

方法一：index-time 的时候生成 term vectors信息

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

"text": {

"type": "text",

"term\_vector": "with\_positions\_offsets\_payloads",

"store" : true,

"analyzer" : "fulltext\_analyzer"

},

"fullname": {

"type": "text",

"analyzer" : "fulltext\_analyzer"

}

}

}

},

"settings" : {

"index" : {

"number\_of\_shards" : 1,

"number\_of\_replicas" : 0

},

"analysis": {

"analyzer": {

"fulltext\_analyzer": {

"type": "custom",

"tokenizer": "whitespace",

"filter": [

"lowercase",

"type\_as\_payload"

]

}

}

}

}

}

添加测试数据：

PUT /my\_index/my\_type/1

{

"fullname" : "Leo Li",

"text" : "hello test test test "

}

PUT /my\_index/my\_type/2

{

"fullname" : "Leo Li",

"text" : "other hello test ..."

}

获取 term vectors 信息：

GET /my\_index/my\_type/1/\_termvectors

{

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

|  |
| --- |
| {  "\_index": "my\_index",  "\_type": "my\_type",  "\_id": "1",  "\_version": 1,  "found": true,  "took": 10,  "term\_vectors": {  "text": {  "field\_statistics": {  "sum\_doc\_freq": 6,  "doc\_count": 2,  "sum\_ttf": 8  },  "terms": {  "hello": {  "doc\_freq": 2,  "ttf": 2,  "term\_freq": 1,  "tokens": [  {  "position": 0,  "start\_offset": 0,  "end\_offset": 5,  "payload": "d29yZA=="  }  ]  },  "test": {  "doc\_freq": 2,  "ttf": 4,  "term\_freq": 3,  "tokens": [  {  "position": 1,  "start\_offset": 6,  "end\_offset": 10,  "payload": "d29yZA=="  },  {  "position": 2,  "start\_offset": 11,  "end\_offset": 15,  "payload": "d29yZA=="  },  {  "position": 3,  "start\_offset": 16,  "end\_offset": 20,  "payload": "d29yZA=="  }  ]  }  }  }  }  } |

3、query-time term vector实验

GET /my\_index/my\_type/1/\_termvectors

{

"fields" : ["fullname"],

"offsets" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

一般来说，如果条件允许，你就用query time的term vector就可以了，你要探查什么数据，现场去探查一下就好了

4、手动指定doc的term vector

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true

}

手动指定一个doc，实际上不是要指定doc，而是要指定你想要安插的词条，hello test，那么就可以放在一个field中

将这些term分词，然后对每个term，都去计算它在现有的所有doc中的一些统计信息

这个挺有用的，可以让你手动指定要探查的term的数据情况，你就可以指定探查“大话西游”这个词条的统计信息

5、手动指定analyzer来生成term vector

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true,

"per\_field\_analyzer" : {

"text": "standard"

}

}

6、terms filter

GET /my\_index/my\_type/\_termvectors

{

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

},

"fields" : ["text"],

"offsets" : true,

"payloads" : true,

"positions" : true,

"term\_statistics" : true,

"field\_statistics" : true,

"filter": {

"max\_num\_terms" : 3,

"min\_term\_freq" : 1,

"min\_doc\_freq" : 1

}

}

这个就是说，根据term统计信息，过滤出你想要看到的term vector统计结果

也挺有用的，比如你探查数据把，可以过滤掉一些出现频率过低的term，就不考虑了

7、multi term vector

GET \_mtermvectors

{

"docs": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "2",

"term\_statistics": true

},

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"fields": [

"text"

]

}

]

}

GET /my\_index/\_mtermvectors

{

"docs": [

{

"\_type": "test",

"\_id": "2",

"fields": [

"text"

],

"term\_statistics": true

},

{

"\_type": "test",

"\_id": "1"

}

]

}

GET /my\_index/my\_type/\_mtermvectors

{

"docs": [

{

"\_id": "2",

"fields": [

"text"

],

"term\_statistics": true

},

{

"\_id": "1"

}

]

}

GET /\_mtermvectors

{

"docs": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"doc" : {

"fullname" : "Leo Li",

"text" : "hello test test test"

}

},

{

"\_index": "my\_index",

"\_type": "my\_type",

"doc" : {

"fullname" : "Leo Li",

"text" : "other hello test ..."

}

}

]

}

### 深入剖析搜索结果的 highlight 高亮显示

大纲：

1、一个最基本的高亮例子

2、三种highlight介绍

3、设置高亮html标签，默认是<em>标签

1、一个最基本的高亮例子

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

PUT /blog\_website/blogs/1

{

"title": "我的第一篇博客",

"content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"

}

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"title": "博客"

}

},

"highlight": {

"fields": {

"title": {}

}

}

}

结果：

|  |
| --- |
| {  "took": 103,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 0.28582606,  "hits": [  {  "\_index": "blog\_website",  "\_type": "blogs",  "\_id": "1",  "\_score": 0.28582606,  "\_source": {  "title": "我的第一篇博客",  "content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"  },  "highlight": {  "title": [  "我的第一篇<em>博客</em>"  ]  }  }  ]  }  } |

<em></em>表现，会变成红色，所以说你的指定的field中，如果包含了那个搜索词的话，就会在那个field的文本中，对搜索词进行红色的高亮显示

GET /blog\_website/blogs/\_search

{

"query": {

"bool": {

"should": [

{

"match": {

"title": "博客"

}

},

{

"match": {

"content": "博客"

}

}

]

}

},

"highlight": {

"fields": {

"title": {},

"content": {}

}

}

}

highlight中的field，必须跟query中的field一一对齐的

2、三种highlight介绍

这里是针对2.x的版本介绍的，6.x 版本已经修改了

①：plain highlight，底层使用lucene highlight，也是 ES默认的分词器

②：posting highlight，index\_options=offsets

（1）性能比plain highlight要高，因为不需要重新对高亮文本进行分词

（2）对磁盘的消耗更少

（3）将文本切割为句子，并且对句子进行高亮，效果更好

PS：在 5.6 的文本中有如下的例子：

This highlighter type is deprecated and will be removed in 6.0. Please use the unified highlighter instead.（https://www.elastic.co/guide/en/elasticsearch/reference/5.6/search-request-highlighting.html）

下面举个例子：

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word",

"index\_options": "offsets"

}

}

}

}

}

PUT /blog\_website/blogs/1

{

"title": "我的第一篇博客",

"content": "大家好，这是我写的第一篇博客，特别喜欢这个博客网站！！！"

}

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"fields": {

"content": {}

}

}

}

③：fast vector highlight

index-time term vector设置在mapping中，就会用fast verctor highlight

（1）对大field而言（大于1mb），性能更高

举例子：

PUT /blog\_website

{

"mappings": {

"blogs": {

"properties": {

"title": {

"type": "text",

"analyzer": "ik\_max\_word"

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word",

"term\_vector" : "with\_positions\_offsets"

}

}

}

}

}

强制使用某种highlighter，比如对于开启了term vector的field而言，可以强制使用plain highlight

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"fields": {

"content": {

"type": "plain"

}

}

}

}

总结一下，其实可以根据你的实际情况去考虑，一般情况下，用plain highlight也就足够了，不需要做其他额外的设置

如果对高亮的性能要求很高，可以尝试启用posting highlight

如果field的值特别大，超过了1M，那么可以用fast vector highlight

3、设置高亮html标签，默认是<em>标签

GET /blog\_website/blogs/\_search

{

"query": {

"match": {

"content": "博客"

}

},

"highlight": {

"pre\_tags": ["<tag1>"],

"post\_tags": ["</tag2>"],

"fields": {

"content": {

"type": "plain"

}

}

}

}

4、高亮片段fragment的设置

GET /\_search

{

"query" : {

"match": { "user": "kimchy" }

},

"highlight" : {

"fields" : {

"content" : {"fragment\_size" : 150, "number\_of\_fragments" : 3, "no\_match\_size": 150 }

}

}

}

fragment\_size: 你一个Field的值，比如有长度是1万，但是你不可能在页面上显示这么长啊。。。设置要显示出来的fragment文本判断的长度，默认是100

number\_of\_fragments：你可能你的高亮的fragment文本片段有多个片段，你可以指定就显示几个片段

no\_match\_size：如果说，对于那些你的 query 没有匹配到的 doc，你的高亮可以显示前缀多少个字符的文本

### 使用 search template 将搜索模板优化

搜索模板，search template，高级功能，就可以将我们的一些搜索进行模板化，然后的话，每次执行这个搜索，就直接调用模板，给传入一些参数就可以了

越高级的功能，越少使用，可能只有在你真的遇到特别合适的场景的时候，才会去使用某个高级功能。但是，这些高级功能你是否掌握，其实就是普通的es开发人员，和es高手之间的一个区别。高手，一般来说，会把一个技术掌握的特别好，特别全面，特别深入，也许他平时用不到这个技术，但是当真的遇到一定的场景的时候，高手可以基于自己的深厚的技术储备，立即反应过来，找到一个合适的解决方案。

如果是一个普通的技术人员，一般只会学习跟自己当前工作相关的一些知识和技术，只要求自己掌握的技术可以解决工作目前遇到的问题就可以了，就满足了，就会止步不前了，然后就不会更加深入的去学习一个技术。但是，当你的项目真正遇到问题的时候，遇到了一些难题，你之前的那一点技术储备已经没法去应付这些更加困难的问题了，此时，普通的技术人员就会扎耳挠腮，没有任何办法。

高手，对技术是很有追求，能够精通很多自己遇到过的技术，但是也许自己学的很多东西，自己都没用过，但是不要紧，这是你的一种技术储备。

1、search template入门

GET /blog\_website/blogs/\_search/template

{

"inline" : {

"query": {

"match" : {

**"{{field}}" : "{{value}}"**

}

}

},

"params" : {

"field" : "title",

"value" : "博客"

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客"

}

}

}

search template："{{field}}" : "{{value}}"

2、toJson

GET /blog\_website/blogs/\_search/template

{

"inline": "{\"query\": {\"match\": {{#toJson}}matchCondition{{/toJson}}}}",

"params": {

"matchCondition": {

"title": "博客"

}

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客"

}

}

}

3、join

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"match": {

"title": "{{#join delimiter=' '}}titles{{/join delimiter=' '}}"

}

}

},

"params": {

"titles": ["博客", "网站"]

}

}

博客,网站

GET /blog\_website/blogs/\_search

{

"query": {

"match" : {

"title" : "博客 网站"

}

}

}

4、default value

POST /blog\_website/blogs/1/\_update

{

"doc": {

"views": 5

}

}

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"range": {

"views": {

"gte": "{{start}}",

"lte": "{{end}}{{^end}}20{{/end}}"

}

}

}

},

"params": {

"start": 1,

"end": 10

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"range": {

"views": {

"gte": 1,

"lte": 10

}

}

}

}

GET /blog\_website/blogs/\_search/template

{

"inline": {

"query": {

"range": {

"views": {

"gte": "{{start}}",

"lte": "{{end}}{{^end}}20{{/end}}"

}

}

}

},

"params": {

"start": 1

}

}

GET /blog\_website/blogs/\_search

{

"query": {

"range": {

"views": {

"gte": 1,

"lte": 20

}

}

}

}

5、conditional

es的config/scripts目录下，预先保存这个复杂的模板，后缀名是.mustache，文件名是conditonal

{

"query": {

"bool": {

"must": {

"match": {

"line": "{{text}}"

}

},

"filter": {

{{#line\_no}}

"range": {

"line\_no": {

{{#start}}

"gte": "{{start}}"

{{#end}},{{/end}}

{{/start}}

{{#end}}

"lte": "{{end}}"

{{/end}}

}

}

{{/line\_no}}

}

}

}

}

GET /my\_index/my\_type/\_search

{

"took": 4,

"timed\_out": false,

"\_shards": {

"total": 5,

"successful": 5,

"failed": 0

},

"hits": {

"total": 1,

"max\_score": 1,

"hits": [

{

"\_index": "my\_index",

"\_type": "my\_type",

"\_id": "1",

"\_score": 1,

"\_source": {

"line": "我的博客",

"line\_no": 5

}

}

]

}

}

GET /my\_index/my\_type/\_search/template

{

"file": "conditional",

"params": {

"text": "博客",

"line\_no": true,

"start": 1,

"end": 10

}

}

6、保存search template

config/scripts，.mustache

提供一个思路

比如说，一般在大型的团队中，可能不同的人，都会想要执行一些类似的搜索操作

这个时候，有一些负责底层运维的一些同学，就可以基于search template，封装一些模板出来，然后是放在各个es进程的scripts目录下的

其他的团队，其实就不用各个团队自己反复手写复杂的通用的查询语句了，直接调用某个搜索模板，传入一些参数就好了

### 基于completion suggest实现搜索提示

suggest，completion suggest，自动完成，搜索推荐，搜索提示 --> 自动完成，auto completion

auto completion

比如说我们在百度，搜索，你现在搜索“大话西游” -->

百度，自动给你提示，“大话西游电影”，“大话西游小说”， “大话西游手游”

不用你把所有你想要输入的文本都输入完，搜索引擎会自动提示你可能是你想要搜索的那个文本

PUT /news\_website

{

"mappings": {

"news" : {

"properties" : {

"title" : {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"suggest" : {

"type" : "completion",

"analyzer": "ik\_max\_word"

}

}

},

"content": {

"type": "text",

"analyzer": "ik\_max\_word"

}

}

}

}

}

completion：es实现的时候，是非常高性能的，会构建不是倒排索引，也不是正排索引，就是纯的用于进行前缀搜索的一种特殊的数据结构，而且会全部放在内存中，所以auto completion进行的前缀搜索提示，性能是非常高的

大话西游

先添加一些数据：

PUT /news\_website/news/1

{

"title": "大话西游电影",

"content": "大话西游的电影时隔20年即将在2017年4月重映"

}

PUT /news\_website/news/2

{

"title": "大话西游小说",

"content": "某知名网络小说作家已经完成了大话西游同名小说的出版"

}

PUT /news\_website/news/3

{

"title": "大话西游手游",

"content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"

}

然后进行搜索：

GET /news\_website/news/\_search

{

"suggest": {

"my-suggest" : {

"prefix" : "大话西游",

"completion" : {

"field" : "title.suggest"

}

}

}

}

|  |
| --- |
| {  "took": 6,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 0,  "max\_score": 0,  "hits": []  },  "suggest": {  "my-suggest": [  {  "text": "大话西游",  "offset": 0,  "length": 4,  "options": [  {  "text": "大话西游小说",  "\_index": "news\_website",  "\_type": "news",  "\_id": "2",  "\_score": 1,  "\_source": {  "title": "大话西游小说",  "content": "某知名网络小说作家已经完成了大话西游同名小说的出版"  }  },  {  "text": "大话西游手游",  "\_index": "news\_website",  "\_type": "news",  "\_id": "3",  "\_score": 1,  "\_source": {  "title": "大话西游手游",  "content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"  }  },  {  "text": "大话西游电影",  "\_index": "news\_website",  "\_type": "news",  "\_id": "1",  "\_score": 1,  "\_source": {  "title": "大话西游电影",  "content": "大话西游的电影时隔20年即将在2017年4月重映"  }  }  ]  }  ]  }  } |

GET /news\_website/news/\_search

{

"query": {

"match": {

"content": "大话西游电影"

}

}

}

|  |
| --- |
| {  "took": 9,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 3,  "max\_score": 1.3495269,  "hits": [  {  "\_index": "news\_website",  "\_type": "news",  "\_id": "1",  "\_score": 1.3495269,  "\_source": {  "title": "大话西游电影",  "content": "大话西游的电影时隔20年即将在2017年4月重映"  }  },  {  "\_index": "news\_website",  "\_type": "news",  "\_id": "3",  "\_score": 1.217097,  "\_source": {  "title": "大话西游手游",  "content": "网易游戏近日出品了大话西游经典IP的手游，正在火爆内测中"  }  },  {  "\_index": "news\_website",  "\_type": "news",  "\_id": "2",  "\_score": 1.1299736,  "\_source": {  "title": "大话西游小说",  "content": "某知名网络小说作家已经完成了大话西游同名小说的出版"  }  }  ]  }  } |

### 使用dynamic mapping模板定制自己的映射策略

高级的用法

比如说，我们本来没有某个type，或者没有某个field，但是希望在插入数据的时候，es自动为我们做一个识别，动态映射出这个type的mapping，包括每个field的数据类型，一般用的动态映射，dynamic mapping

这里有个问题，如果说，我们其实对dynamic mapping有一些自己独特的需求，比如说，es默认来说，如经过识别到一个数字，field: 10，默认是搞成这个field的数据类型是long，再比如说，如果我们弄了一个field : "10"，默认就是text，还会带一个keyword的内置field。我们没法改变。

但是我们现在就是希望动态映射的时候，根据我们的需求去映射，而不是让es自己按照默认的规则去玩儿

这个时候就可以用：dyanmic mapping template，动态映射模板

我们自己预先定义一个模板，然后插入数据的时候，相关的field，如果能够根据我们预先定义的规则，匹配上某个我们预定义的模板，那么就会根据我们的模板来进行mapping，决定这个Field的数据类型

\*\* 默认的动态映射的效果咋样？

DELETE /my\_index

PUT /my\_index/my\_type/1

{

"test\_string": "hello world",

"test\_number": 10

}

es的自动的默认的，动态映射是咋样的。。。

GET /my\_index/\_mapping/my\_type

{

"my\_index": {

"mappings": {

"my\_type": {

"properties": {

"test\_number": {

"type": "long"

},

"test\_string": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

这个就是es的默认的动态映射规则，可能就不是我们想要的。。。

我们比如说，现在想要的效果是啥。。。

test\_number，如果是个数字，我们希望默认就是integer类型的

test\_string，如果是字符串，我们希望默认是个text，这个没问题，但是内置的field名字，叫做raw，不叫座keyword，类型还是keyword，保留500个字符

动态映射模板，有两种方式，

第一种，是根据新加入的field的默认的数据类型，来进行匹配，匹配上某个预定义的模板；

第二种，是根据新加入的field的名字，去匹配预定义的名字，或者去匹配一个预定义的通配符，然后匹配上某个预定义的模板

下面详细看下面的两种方式的例子：

1、根据类型匹配映射模板

PUT my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"integers": {

"match\_mapping\_type": "long",

"mapping": {

"type": "integer"

}

}

},

{

"strings": {

"match\_mapping\_type": "string",

"mapping": {

"type": "text",

"fields": {

"raw": {

"type": "keyword",

"ignore\_above": 500

}

}

}

}

}

]

}

}

}

PUT /my\_index/my\_type/1

{

"test\_long": 1,

"test\_string": "hello world"

}

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"integers": {

"match\_mapping\_type": "long",

"mapping": {

"type": "integer"

}

}

},

{

"strings": {

"match\_mapping\_type": "string",

"mapping": {

"fields": {

"raw": {

"ignore\_above": 500,

"type": "keyword"

}

},

"type": "text"

}

}

}

],

"properties": {

"test\_number": {

"type": "integer"

},

"test\_string": {

"type": "text",

"fields": {

"raw": {

"type": "keyword",

"ignore\_above": 500

}

}

}

}

}

}

}

2、根据字段名配映射模板

PUT /my\_index

{

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"string\_as\_integer": {

"match\_mapping\_type": "string",

"match": "long\_\*",

"unmatch": "\*\_text",

"mapping": {

"type": "integer"

}

}

}

]

}

}

}

举个例子，field : "10"，把类似这种field，弄成long型

{

"my\_index": {

"mappings": {

"my\_type": {

"dynamic\_templates": [

{

"string\_as\_integer": {

"match": "long\_\*",

"unmatch": "\*\_text",

"match\_mapping\_type": "string",

"mapping": {

"type": "integer"

}

}

}

],

"properties": {

"long\_field": {

"type": "integer"

},

"long\_field\_text": {

"type": "text",

"fields": {

"keyword": {

"type": "keyword",

"ignore\_above": 256

}

}

}

}

}

}

}

}

场景：有些时候，dynamic mapping + template，每天有一堆日志，每天有一堆数据

这些数据，每天的数据都放一个新的type中，每天的数据都会哗哗的往新的tye中写入，此时你就可以定义一个模板，搞一个脚本，每天都预先生成一个新type的模板，里面讲你的各个Field都匹配到一个你预定义的模板中去，就好了

### 地理位置-学习使用 geo point 地理位置数据类型

这一讲开始，后面会跟着几讲内容，将地理位置相关的知识给大家讲解一下

主要是es支持基于地理位置的搜索，和聚合分析的

举个例子，比如说，我们后面就会给大家演示一下，你现在如果说做了一个酒店o2o app，让你的用户在任何地方，都可以根据当前所在的位置，找到自己身边的符合条件的一些酒店，那么此时就完全可以使用es来实现，非常合适

我现在在上海某个大厦附近，我要搜索到距离我2公里以内的5星级的带游泳池的一个酒店s，用es就完全可以实现类似这样的基于地理位置的搜索引擎

1、建立geo\_point类型的mapping

第一个地理位置的数据类型，就是geo\_point，geo\_point，说白了，就是一个地理位置坐标点，包含了一个经度，一个维度，经纬度，就可以唯一定位一个地球上的坐标

PUT /my\_index

{

"mappings": {

"my\_type": {

"properties": {

**"location"**: {

"type": "**geo\_point**"

}

}

}

}

}

2、写入geo\_point的3种方法

方法一：

PUT my\_index/my\_type/1

{

"text": "Geo-point as an object",

"location": {

"lat": 41.12,

"lon": -71.34

}

}

latitude：维度

longitude：经度

我们这里就不用去关心，这些坐标到底代表什么地方，其实都是我自己随便写的，只要能够作为课程，给大家演示清楚就可以了，自己去找一些提供地理位置的一些公司，供应商，api，百度地图，也是提供各个地方的经纬度的

不建议用下面两种语法：

方法二：

PUT my\_index/my\_type/2

{

"text": "Geo-point as a string",

"location": "41.12,-71.34"

}

方法三：

PUT my\_index/my\_type/4

{

"text": "Geo-point as an array",

"location": [ -71.34, 41.12 ]

}

3、根据地理位置进行查询

最最简单的，根据地理位置查询一些点，比如说，下面geo\_bounding\_box查询，查询某个矩形的地理位置范围内的坐标点

GET /my\_index/my\_type/\_search

{

"query": {

"geo\_bounding\_box": {

"location": {

"top\_left": {

"lat": 42,

"lon": -72

},

"bottom\_right": {

"lat": 40,

"lon": -74

}

}

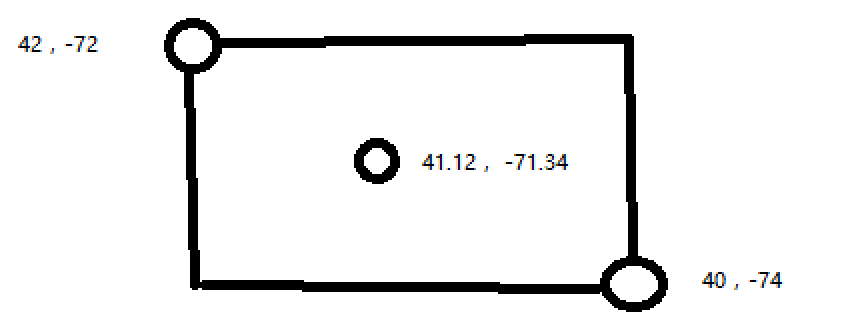
}

}

}

|  |
| --- |
| {  "took": 81,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 1,  "hits": [  {  "\_index": "my\_index",  "\_type": "my\_type",  "\_id": "1",  "\_score": 1,  "\_source": {  "location": {  "lat": 41.12,  "lon": -71.34  }  }  }  ]  }  } |

比如41.12,-71.34就是一个酒店，然后我们现在搜索的是从42,-72（代表了大厦A）和40,-74（代表了马路B）作为矩形的范围，在这个范围内的酒店，是什么



### 地理位置-酒店020搜索案例以及搜索指定区域内的酒店

稍微真实点的案例，酒店o2o app作为一个背景，用各种各样的方式，去搜索你当前所在的地理位置附近的酒店

搜索指定区域范围内的酒店，比如说，我们可以在搜索的时候，指定两个地点，就要在东方明珠大厦和上海路组成的矩阵的范围内，搜索我想要的酒店

PUT /hotel\_app

{

"mappings": {

"hotels": {

"properties": {

"pin": {

"properties": {

"location": {

**"type": "geo\_point"**

}

}

}

}

}

}

}

PUT /hotel\_app/hotels/1

{

"name": "喜来登大酒店",

"pin" : {

"location" : {

"lat" : 40.12,

"lon" : -71.34

}

}

}

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

**"geo\_bounding\_box"**: {

"pin.location": {

"top\_left" : {

"lat" : 40.73,

"lon" : -74.1

},

"bottom\_right" : {

"lat" : 40.01,

"lon" : -71.12

}

}

}

}

}

}

}

我们现在要指定东方明珠大厦，上海路，上海博物馆，这三个地区组成的多边形的范围内，我要搜索这里面的酒店

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

"geo\_polygon": {

"pin.location": {

"points": [

{"lat" : 40.73, "lon" : -74.1},

{"lat" : 40.01, "lon" : -71.12},

{"lat" : 50.56, "lon" : -90.58}

]

}

}

}

}

}

}

### 地理位置-实战搜索距离当前位置一定范围内的酒店

酒店o2o app，作为案例背景

比如说，现在用户，所在的位置，是个地理位置的坐标，我是知道我的坐标的，app是知道的，android，地理位置api，都可以拿到当前手机app的经纬度

我说，我现在就要搜索出，举例我200m，或者1公里内的酒店

重要！！！！

我们之前出去玩儿，都会用一些酒店o2o app，典型的代表，很多旅游app，一般来说，我们怎么搜索，到了一个地方，就会搜索说，我这里举例几百米，2公里内的酒店，搜索一下

上节课讲解的，其实也很重要，一般来说，发生在我们在去旅游之前，会现在旅游app上搜索一个区域内的酒店，比如说，指定了西安火车站、西安博物馆，拿指定的几个地方的地理位置，组成一个多边形区域范围，去搜索这个区域内的酒店

承认，一些案例，当然不可能说达到讲解真实的复杂的大型的项目的效果来的那么好，光是学知识，学技术而言，有一些案例就非常不错了

后面，会讲解真正的企业级的大型的搜索引擎，真实复杂业务的数据分析系统的项目

GET /hotel\_app/hotels/\_search

{

"query": {

"bool": {

"must": [

{

"match\_all": {}

}

],

"filter": {

**"geo\_distance":** {

"distance": "200km",

"pin.location": {

"lat": 40,

"lon": -70

}

}

}

}

}

}

### 地理位置-统计当前位置每个距离范围内有多少家酒店

最后一个知识点，基于地理位置进行聚合分析

我的需求就是，统计一下，举例我当前坐标的几个范围内的酒店的数量，比如说举例我0~100m有几个酒店，100m~300m有几个酒店，300m以上有几个酒店

一般来说，做酒店app，一般来说，我们是不是会有一个地图，用户可以在地图上直接查看和搜索酒店，此时就可以显示出来举例你当前的位置，几个举例范围内，有多少家酒店，让用户知道，心里清楚，用户体验就比较好

GET /hotel\_app/hotels/\_search

{

"size": 0,

"aggs": {

"agg\_by\_distance\_range": {

**"geo\_distance":** {

"field": "pin.location",

"origin": {

"lat": 40,

"lon": -70

},

"unit": "mi",

"ranges": [

{

"to": 100

},

{

"from": 100,

"to": 300

},

{

"from": 300

}

]

}

}

}

}

结果：

|  |
| --- |
| {  "took": 5,  "timed\_out": false,  "\_shards": {  "total": 5,  "successful": 5,  "failed": 0  },  "hits": {  "total": 1,  "max\_score": 0,  "hits": []  },  "aggregations": {  "agg\_by\_distance\_range": {  "buckets": [  {  "key": "\*-100.0",  "from": 0,  "to": 100,  "doc\_count": 1  },  {  "key": "100.0-300.0",  "from": 100,  "to": 300,  "doc\_count": 0  },  {  "key": "300.0-\*",  "from": 300,  "doc\_count": 0  }  ]  }  }  } |

m (metres) but it can also accept: m (miles), km (kilometers)

sloppy\_arc (the default), arc (most accurate) and plane (fastest)

## 熟练掌握 ES-JavaAPI

### Client集群自动探查以及汽车零店案例背景

在快速入门篇的时候，讲解过了一些基本的java api，包括了document增删改查，基本的搜索，基本的聚合

高手进阶篇，必须将java api这块深入讲解一下，介绍一些最常用的，最核心的一些api的使用，用一个模拟现实的案例背景，让大家在学习的时候更加贴近业务

话说在前面，我们是不可能将所有的java api用视频全部录制一遍的，因为api太多了。。。。

我们之前讲解各种功能，各种知识点，花了那么多的时间，哪儿些才是最最关键的，知识，原理，功能，es restful api，最次最次，哪怕是搞php，搞python的人也可以来学习

如果说，现在要将所有所有的api全部用java api实现一遍和讲解，太耗费时间了，几乎不可能接受

采取的粗略，将核心的java api语法，还有最最常用的那些api都给大家上课演示了

然后最后一讲，会告诉大家，在掌握了之前那些课程讲解的各种知识点之后，如果要用java api去实现和开发，应该怎么自己去探索和掌握

java api，api的学习，实际上是最最简单的，纯用，没什么难度，技术难度，你掌握了课上讲解的这些api之后，自己应该就可以举一反三，后面自己去探索和尝试出自己要用的各种功能对应的java api是什么。。。

1、client集群自动探查

默认情况下，是根据我们手动指定的所有节点，依次轮询这些节点，来发送各种请求的，如下面的代码，我们可以手动为client指定多个节点

TransportClient client = new PreBuiltTransportClient(settings)

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost1"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost2"), 9300))

.addTransportAddress(new InetSocketTransportAddress(InetAddress.getByName("localhost3"), 9300));

但是问题是，如果我们有成百上千个节点呢？难道也要这样手动添加吗？

es client提供了一种集群节点自动探查的功能，打开这个自动探查机制以后，es client会根据我们手动指定的几个节点连接过去，然后通过集群状态自动获取当前集群中的所有data node，然后用这份完整的列表更新自己内部要发送请求的node list。默认每隔5秒钟，就会更新一次node list。

但是注意，es cilent是不会将Master node纳入node list的，因为要避免给master node发送搜索等请求。

这样的话，我们其实直接就指定几个master node，或者1个node就好了，client会自动去探查集群的所有节点，而且每隔5秒还会自动刷新。非常棒。

Settings settings = Settings.builder()

.put(**"client.transport.sniff", true**).build();

TransportClient client = new PreBuiltTransportClient(settings);

使用上述的settings配置，将client.transport.sniff设置为true即可打开集群节点自动探查功能

在实际的生产环境中，都是这么玩儿的。。。

2、汽车零售案例背景

简单来说，会涉及到三个数据，**汽车信息，汽车销售记录，汽车4S店信息**

### 基于 upsert 实现汽车最新价格的调整

做一个汽车零售数据的mapping，我们要做的第一份数据，其实汽车信息

PUT /car\_shop

{

"mappings": {

"cars": {

"properties": {

"brand": {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"raw": {

"type": "keyword"

}

}

},

"name": {

"type": "text",

"analyzer": "ik\_max\_word",

"fields": {

"raw": {

"type": "keyword"

}

}

}

}

}

}

}

首先的话呢，第一次调整宝马320这个汽车的售价，我们希望将售价设置为32万，用一个upsert语法，如果这个汽车的信息之前不存在，那么就insert，如果存在，那么就update

|  |
| --- |
| // 添加数据  IndexRequest indexRequest = new IndexRequest("car\_shop", "cars", "1")  .source(XContentFactory.jsonBuilder()  .startObject()  .field("brand", "宝马")  .field("name", "宝马320")  .field("price", 320000)  .field("produce\_date", "2017-01-01")  .endObject());  // 更新的数据  UpdateRequest updateRequest = new UpdateRequest("car\_shop", "cars", "1")  .doc(XContentFactory.jsonBuilder()  .startObject()  .field("price", 320000)  .endObject())  .upsert(indexRequest); // 绑定 IndexRequest    // client 更新请求  client.update(updateRequest).get(); |
| IndexRequest indexRequest = new IndexRequest("car\_shop", "cars", "1")  .source(jsonBuilder()  .startObject()  .field("brand", "宝马")  .field("name", "宝马320")  .field("price", 310000)  .field("produce\_date", "2017-01-01")  .endObject());  UpdateRequest updateRequest = new UpdateRequest("car\_shop", "cars", "1")  .doc(jsonBuilder()  .startObject()  .field("price", 310000)  .endObject())  .upsert(indexRequest);  client.update(updateRequest).get(); |

### 基于 mget 实现多俩汽车的配置与价格对比

场景，一般来说，我们都可以在一些汽车网站上，或者在混合销售多个品牌的汽车4S店的内部，都可以在系统里调出来多个汽车的信息，放在网页上，进行对比

mget，一次性将多个document的数据查询出来，放在一起显示，多个汽车的型号，一次性拿出了多辆汽车的信息

PUT /car\_shop/cars/2

{

"brand": "奔驰",

"name": "奔驰C200",

"price": 350000,

"produce\_date": "2017-01-05"

}

|  |
| --- |
| MultiGetResponse multiGetItemResponses = client.prepareMultiGet()  .add("car\_shop", "cars", "1")  .add("car\_shop", "cars", "2")  .get();  for (MultiGetItemResponse itemResponse : multiGetItemResponses) {  GetResponse response = itemResponse.getResponse();  if (response.isExists()) {  String json = response.getSourceAsString();  }  } |

### 基于 bulk 实现多 4s 点销售数据批量上传

业务场景：有一个汽车销售公司，拥有很多家4S店，这些4S店的数据，都会在一段时间内陆续传递过来，汽车的销售数据，现在希望能够在内存中缓存比如1000条销售数据，然后一次性批量上传到es中去

PUT /car\_shop/sales/1

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 300000,

"sale\_date": "2017-01-21"

}

PUT /car\_shop/sales/2

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 300000,

"sale\_date": "2017-01-21"

}

|  |
| --- |
| BulkRequestBuilder bulkRequest = client.prepareBulk();  bulkRequest.add(client.prepareIndex("car\_shop", "sales", "3")  .setSource(jsonBuilder()  .startObject()  .field("brand", "奔驰")  .field("name", "奔驰C200")  .field("price", 350000)  .field("produce\_date", "2017-01-05")  .field("sale\_price", 340000)  .field("sale\_date", "2017-02-03")  .endObject()  )  );  bulkRequest.add(client.prepareUpdate("car\_shop", "sales", "1")  .setDoc(jsonBuilder()  .startObject()  .field("sale\_price", "290000")  .endObject()  )  );  bulkRequest.add(client.prepareDelete("car\_shop", "sales", "2"));  BulkResponse bulkResponse = bulkRequest.get();  if (bulkResponse.hasFailures()) {} |

### 基于 scroll 实现阅读销售数据批量下降

比如说，现在要下载大批量的数据，从es，放到excel中，我们说，月度，或者年度，销售记录，很多，比如几千条，几万条，几十万条

其实就要用到我们之前讲解的es scroll api，对大量数据批量的获取和处理

PUT /car\_shop/sales/4

{

"brand": "宝马",

"name": "宝马320",

"price": 320000,

"produce\_date": "2017-01-01",

"sale\_price": 280000,

"sale\_date": "2017-01-25"

}

就是要看宝马的销售记录

2条数据，做一个演示，每个批次下载一条宝马的销售记录，分2个批次给它下载完

|  |
| --- |
| SearchResponse scrollResp = client.prepareSearch("car\_shop")  .addTypes("sales")  .setScroll(new TimeValue(60000))  .setQuery(QueryBuilders.termQuery("brand.raw", "宝马"))  .setSize(1)  .get();  do {  for (SearchHit hit : scrollResp.getHits().getHits()) {    }    scrollResp = client.prepareSearchScroll(scrollResp.getScrollId())  .setScroll(new TimeValue(60000))  .execute()  .actionGet();  } while(scrollResp.getHits().getHits().length != 0); |

### 基于 search template 实现按品牌分页查询模板

搜索模板的功能，java api怎么去调用一个搜索模板

模板放在：{ES\_HOME}/config/scripts/ 目录下

模板名称：page\_query\_by\_brand.mustache

模板内容：

{

"from": {{from}},

"size": {{size}},

"query": {

"match": {

"brand.keyword": "{{brand}}"

}

}

}

|  |
| --- |
| Map<String, Object> params = new HashMap<>(); params.put("from", 0); params.put("size", 1); params.put("brand", "宝马");  SearchResponse response = new SearchTemplateRequestBuilder(client)  .setScript("page\_query\_by\_brand")  .setScriptType(ScriptType.*FILE*)  .setScriptParams(params)  .setRequest(new SearchRequest("car\_shop").types("sales"))  .get()  .getResponse(); |

### 对汽车品牌进行全文检索、精确查询和前缀搜索

PUT /car\_shop/cars/5

{

"brand": "华晨宝马",

"name": "宝马318",

"price": 270000,

"produce\_date": "2017-01-20"

}

|  |
| --- |
| SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("cars")  .setQuery(QueryBuilders.matchQuery("brand", "宝马"))  .get();  SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("cars")  .setQuery(QueryBuilders.multiMatchQuery("宝马", "brand", "name"))  .get();  SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("cars")  .setQuery(QueryBuilders.commonTermsQuery("name", "宝马320"))  .get();  SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("cars")  .setQuery(QueryBuilders.prefixQuery("name", "宝"))  .get(); |

### 对汽车品牌进行多种条件的组合搜索

课程大纲

|  |
| --- |
| QueryBuilder qb = QueryBuilders.boolQuery()  .must(matchQuery("brand", "宝马"))  .mustNot(termQuery("name.raw", "宝马318"))  .should(termQuery("produce\_date", "2017-01-02"))  .filter(rangeQuery("price").gte("280000").lt("350000"));  SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("cars")  .setQuery(qb)  .get(); |

### 基于地理位置对周围汽车 4s 店进行搜索

需要先添加如下的 jar。

|  |
| --- |
| <dependency>  <groupId>org.locationtech.spatial4j</groupId>  <artifactId>spatial4j</artifactId>  <version>0.6</version>  </dependency>  <dependency>  <groupId>com.vividsolutions</groupId>  <artifactId>jts</artifactId>  <version>1.13</version>  <exclusions>  <exclusion>  <groupId>xerces</groupId>  <artifactId>xercesImpl</artifactId>  </exclusion>  </exclusions>  </dependency> |

比如我们有很多的4s店，然后呢给了用户一个app，在某个地方的时候，可以根据当前的地理位置搜索一下，自己附近的4s店

POST /car\_shop/\_mapping/shops

{

"properties": {

"pin": {

"properties": {

"location": {

"type": "geo\_point"

}

}

}

}

}

PUT /car\_shop/shops/1

{

"name": "上海至全宝马4S店",

"pin" : {

"location" : {

"lat" : 40.12,

"lon" : -71.34

}

}

}

第一个需求：搜索两个坐标点组成的一个区域

|  |
| --- |
| QueryBuilder qb = geoBoundingBoxQuery("pin.location").setCorners(40.73, -74.1, 40.01, -71.12); |

第二个需求：指定一个区域，由三个坐标点，组成，比如上海大厦，东方明珠塔，上海火车站

|  |
| --- |
| List<GeoPoint> points = new ArrayList<>();  points.add(new GeoPoint(40.73, -74.1));  points.add(new GeoPoint(40.01, -71.12));  points.add(new GeoPoint(50.56, -90.58));  QueryBuilder qb = geoPolygonQuery("pin.location", points); |

第三个需求：搜索距离当前位置在200公里内的4s店

|  |
| --- |
| QueryBuilder qb = geoDistanceQuery("pin.location").point(40, -70).distance(200, DistanceUnit.KILOMETERS);  SearchResponse response = client.prepareSearch("car\_shop")  .setTypes("shops")  .setQuery(qb)  .get(); |

### 如何自己尝试 API 以掌握所有搜索和聚合的语法

1、自己要什么query，自己去用QueryBuilders去尝试，disMaxQuery

2、自己的某种query，有一些特殊的参数，tieBreaker，自己拿着query去尝试，点一下，看ide的自动提示，自己扫一眼，不就知道有哪些query，哪些参数了

3、如果不是query，是聚合，也是一样的，你就拿AggregationBuilders，点一下，看一下ide的自动提示，我们讲过的各种语法，都有了

4、包括各种聚合的参数，也是一样的，找到对应的AggregationBuilder之后，自己点一下，找需要的参数就可以了

5、自己不断尝试，就o了，组装了一个搜索，或者聚合，自己试一下，测一下

如果你实在找不到，搞不定，就QQ来找我，当然别自己一开始就跑来找我，先自己努力研究一下

有些人说，可以上官网，官网api也没这么全

## 快速入门篇以及高手篇课程总结以及后续课程介绍

**对于基础入门篇的知识，需要达到以下的目的：**

1、快速入门：能了解最最基本的es的一些使用

2、分布式原理：了解es的基本原理

3、分布式文档系统：基本精通es的document相关的一些操作和开发

4、初识搜索引擎：掌握es最核心，最基本的一些搜索的技术

5、索引管理：掌握了基本的es的索引相关的操作

6、内核原理探秘：深入理解的es的底层的原理

7、Java API初步使用：掌握最基础的java api的使用

总结：开始把es用起来，可以玩儿起来，掌握了一些基本的知识，自己在公司做一些最最简单的小项目，也ok

**对于高手进阶篇，需要达到如下目的：**

1、深度探秘搜索技术：彻底深入的了解各种高级搜索技术，精通搜索底层原理

2、彻底掌握IK中文分词器：彻底掌握，连源码的修改都讲过了，怎么基于mysql热加载你的词库

3、深入聚合数据分析：彻底深入的掌握了各种各样的数据聚合分析的功能

4、数据建模实战：对模拟真实世界的有复杂关系的数据模型，讲解了建模、搜索和聚合

5、elasticsearch高手进阶：高级功能，搜索模板，term vector，地理位置的搜索和聚合

6、java api：核心的java api的现场演示，如何自己去摸索所有的java api的使用

你做一些小型的项目，数据量不大，简单在自己公司部署几个节点的es，3个节点

玩儿各种各样的搜索，聚合，中文分词，有关联和层次关系的数据如何建模，document如何管理和操作，索引的基本管理，es的核心原理，java api的系统的使用，高级的功能，基于地理位置的应用的开发

你都能搞定了

你只能做es的小型项目，或者大型项目，但是数据量大不了

============================================================================================================================

**对于后面的两个篇章**

1、运维优化：生产环境的大规模集群的部署、运维管理、监控、安全管理、升级、性能优化、索引管理，大型es集群的运维知识，包括海量数据场景下的性能的调优，还有一个大数据场景的应用系统的设计，范式

搞java的，了解什么es的运维。。。。

你要是搞java的，结果不了解es运维，你也别做es的大型项目，大数据场景下的，你根本就不了解集群，不了解大数据集群环境下的一些特殊的配置，安全，监控，es全景图，概览

你要是搞java，基于es集群，大数据量做开发，你不了解上面这些东西，你碰到了问题，就抓瞎

你如果真是搞java的，最自己的技术有追求，希望自己出去找工作，技术牛逼一些，不要给自己设限制，开发，不要了解运维。如果你是个java架构师，你连es集群相关的知识都不懂，你碰到了问题，你的项目遇到了一些的报错，你都搞不定，你还当什么架构师，或者项目经理

如果你对技术有追求，就好好学一下

2、项目实战：各种各样的案例，作为背景，模拟现实，来用业务驱动课程的讲解，和动手的实战，更好的理解、吸收、刺激你的对技术如何运用的灵感

大型门户网站的搜索引擎系统：安全模拟真实大型搜索引擎系统的架构，特殊的点，降级，防止雪崩，缓存，架构怎么拆分，复杂的搜索引擎怎么构建，讲解

大型电商网站的数据分析系统：完全用真实的复杂的电商的业务场景，去开发一整套完整的涵盖数十个分析模块的es数据分析系统

后面两个篇章才是真正的拔高的地方，如果你对自己技术有追求，想把技术学好一些，建议，前面两个篇章，至少看个2遍，彻底掌握；后面2个篇章可以到时候好好看看

## ES 生产集群部署

### 硬件配置,JVM以及集群规划建议

我们之前一直是在windows环境下去启动一个单节点的es进程，然后去学习和练习各种es的高阶的搜索技术，聚合技术

主要针对的是es的开发，你可以认为是，如果你是一个java工程师的话，然后你们公司已经有人去维护和搭建一个es集群了，那么你直接掌握之前的内容，就足够了

你就可以建立需要的索引，写入数据，搜索，聚合

很多同学，还是需要自己去规划、设计和搭建一个es集群出来，先，然后才能基于es去做一些开发这样的

肯定不是在windows操作系统上去搞了，肯定是在linux集群上面去部署咯

规划一下，比如你需要几台机器，物理机，还是虚拟机呢？每台机器要多大的资源，多少G内存，多少个cpu，多大的磁盘空间，等等，然后对生产环境的机器相关的配置有没有什么特殊的要求，对于es来说。你需要处理多大的数据量？需要多大的集群才能支撑的住？

这块规划好之后，就是搞到一堆你需要的机器，符合你的需求和要求的机器，机器弄好之后，就可以开始在上面部署和玩儿你的es了

是这样的，我们的课程，肯定是只能用虚拟机去演练和模拟，在机器的配置上，能够在课程现场支撑的数据量上，是绝对达不到生产环境的

尽量保证，给大家用虚拟机去还原和模拟一些场景，然后讲解各种相关的技术知识

一般来说，如果我们刚开始用es，都是先在自己的笔记本电脑上，或者是几个虚拟机组成的小集群上，安装一个es，然后开始学习和试用其中的功能。但是如果我们要将es部署到生产环境中，那么是由很多额外的事情要做的。需要考虑我们部署的机器的内存、CPU、磁盘、JVM等各种资源和配置。

1、内存

es是很吃内存的，es吃的主要不是你的jvm的内存，一般来说es用jvm heap（堆内存）还是用的比较少的，主要吃的是你的机器的内存

es底层基于lucene，lucene是基于磁盘文件来读写和保存你的索引数据的，倒排索引，正排索引，lucene的特点就是会基于os filesystem cache，会尽量将频繁访问的磁盘文件的数据在操作系统的内存中进行缓存，然后尽量提升磁盘文件读写的性能

很多同学都问我说，es的性能感觉不太理想，es的性能80%取决于说，你的机器上，除了分配给jvm heap内存以外，还剩下多少内存，剩下的内存会留给es的磁盘索引文件做缓存，如果os cache能够缓存更多的磁盘文件的数据，索引文件的数据，索引读写的性能都会高很多，特别是检索

但是如果你的大量的索引文件在os cache中放不下，还是停留在磁盘上，那么搜索、聚合的时候大量的都是读写磁盘文件，性能当然低了，一个数量级，ms级，s级

问我，es上千万数据的搜索，要耗费10s，大量读写磁盘了

如果在es生产环境中，哪种资源最容易耗尽，那么就是内存了。排序和聚合都会耗费掉很多内存，所以给es进程分配足够的jvm heap内存是很重要的。除了给jvm heap分配内存，还需要给予足够的内存给os filesystem cache。因为lucene用的数据结构都是给予磁盘的格式，es是通过os cache来进行高性能的磁盘文件读写的。

关于机器的内存相关的知识，后面会有很深入的讲解，这里先简单提一下，一般而言，除非说你的数据量很小，比如就是一些os系统，各种信息管理系统，要做一个内部的检索引擎，几万，几十万，几百万的数据量，对机器的资源配置要求还是蛮低的。一般而言，如果你的数据量过亿，几亿，几十亿。那么其实建议你的每台机器都给64G的内存的量。

如果一个机器有64G的内存，那么是比较理想的状态，但是32GB和16GB的内存也是ok的。具体的内存数量还是根据数据量来决定。但是如果一个机器的内存数量小于8G，那么就不太适合生产环境了，因为我们可能就需要很多小内存的机器来搭建集群。而大于64G的机器也不是很有必要。

笔记本电脑，24G内存（16G+8G），双核，虚拟机4台2核4G

2、CPU

大多数的es集群对于cpu的要求都会比较低一些，因此一台机器有多少个cpu core其实对生产环境的es集群部署相对来说没有那么的重要了，至少没有内存来的重要。当然，肯定是要用多核处理器的，一般来说2个cpu core~8个cpu core都是可以的。

此外，如果要选择是较少的cpu core但是cpu性能很高，还是较多的cpu core但是cpu性能较为一般，那么肯定是选择性能较为一般但是更多的cpu core。因为更多的cpu core可以提供更强的并发处理能力，远比单个cpu性能高带来的效果更加明显。

3、磁盘

对于es的生产环境来说，磁盘是非常重要的，尤其是对那些大量写入的es集群，比如互联网公司将每天的实时日志数据以高并发的速度写入es集群。在服务器上，磁盘是最慢的那个资源，所以对于大量写入的es集群来说，会很容易因为磁盘的读写性能造成整个集群的性能瓶颈。

如果我们能够使用SSD固态硬盘，而不是机械硬盘，那么当然是最好的，SSD的性能比机械硬盘可以高很多倍，可以让es的读写性能都高很多倍。所以，如果公司出的起钱大量使用固态硬盘，那么当然是最好的。

连我的笔记本电脑，都是有100多G的SSD啊，还有1T的机械硬盘

如果我们在用SSD硬盘的化，那么需要检查我们的I/O scheduler，需要正确的配置IO scheduler。当我们将数据写入磁盘时，IO scheduler会决定什么时候数据才会真正的写入磁盘，而不是停留在os cache内存缓冲中。大多数机器上，默认的IO scheduler是cfq，也就是completely fair queuing。

这个scheduler会给每个进程都分配一些时间分片，time slice，然后会优化每个进程的数据如何写入磁盘中，优化的思路主要 是根据磁盘的物理布局来决定如何将数据写入磁盘，进而提升写入磁盘的性能。这是针对机械硬盘做出的优化，因为机械硬盘是一种旋转存储介质，是通过机械旋转磁盘+磁头进行磁盘读写的机制。

但是scheduler的这种默认的执行机制，对于SSD来说是不太高效的，因为SSD跟机械硬盘是不一样的，SSD不涉及到机械磁盘旋转和磁头读取这种传统的读写机制。对于SSD来说，应该用deadline/noop scheduler。deadline scheduler会基于写操作被pending了多长时间来进行写磁盘优化，而noop scheduler就是一个简单的FIFO队列先进先出的机制。

调整io scheduler可以带来很大的性能提升，甚至可以达到数百倍。

如果我们没有办法使用SSD，只能使用机械硬盘，那么至少得尽量正确读写速度最快的磁盘，比如高性能的服务器磁盘。

此外，使用RAID 0也是一种提升磁盘读写速度的高效的方式，无论是对于机械硬盘，还是SSD，都一样。RAID 0也被称之为条带式存储机制，striping，在RAID各种级别中性能是最高的。RAID 0的基本原理，是把连续的数据分散存储到多个磁盘上进行读写，也就是对数据进行条带式存储。这样系统的磁盘读写请求就可以被分散到多个磁盘上并行执行。但是没有必要使用镜像或者RAID的其他模式，因为我们不需要通过RAID来实现数据高可用存储，es的replica副本机制本身已经实现了数据高可用存储。

最后，我们要避免跟网络相关的存储模式，network-attached storage，NAS，比如基于网络的分布式存储模式。虽然很多供应商都说他们的NAS解决方案性能非常高，而且比本地存储的可靠性更高。但是实际上用起来会有很多性能和可靠性上的风险，一般因为网络传输会造成较高的延时，同时还有单点故障的风险。

4、网络

对于es这种分布式系统来说，快速而且可靠的网络是非常的重要的。因为高速网络通信可以让es的节点间通信达到低延时的效果，高带宽可以让shard的移动和恢复，以及分配等操作更加的快速。现代的数据中心的网络对于大多数的集群来说，性能都足够高了。比如千兆网卡，这都是可以的。

但是要避免一个集群横跨多个数据中心，比如异地多机房部署一个集群，因为那样的话跨机房，跨地域的传输会导致网络通信和数据传输性能较差。es集群是一种p2p模式的分布式系统架构，不是master-slave主从分布式系统。在es集群中，所有的node都是相等的，任意两个node间的互相通信都是很频繁和正常的。因此如果部署在异地多机房，那么可能会导致node间频繁跨地域进行通信，通信延时会非常高，甚至造成集群运行频繁不正常。

就跟NAS存储模式一样，很多供应商都说跨地域的多数据中心是非常可靠的，而且低延时的。一般来说，可能的确是这样，但是一旦发生了网络故障，那么集群就完了。通常来说，跨地域多机房部署一个es集群带来的效益，远远低于维护这样的集群所带来的额外成本。

5、自建集群 vs 云部署

现在一般很容易就可以拿到高性能的机器来部署集群：很多高性能的机器可以有上百G的内存资源，还有几十个cpu core。但是同时我们也可以再云供应商上，比如阿里云，租用大量的小资源的虚拟机。那么对于自己购买昂贵高性能服务器自建集群，以及租用云机器来部署，该选择哪种方案呢？

你是自己购买5台，比如说，8核64G的物理机，搭建es集群

或者是，比如说，上阿里云，或者其他的云服务，购买了2核4G，16台，虚拟机，搭建es集群

你上阿里云，也可以买到大资源量的虚拟机，4/8/16核64G

一般来说，对于es集群而言，是建议拥有少数机器，但是每个机器的资源都非常多，尽量避免拥有大量的少资源的虚拟机。因为对于运维和管理来说，管理5个物理机组成的es集群，远远比管理100个虚拟机组成的es集群要简单的多。

同时即使是自建集群，也要尽量避免那种超大资源量的超级服务器，因为那样可能造成资源无法完全利用，然后在一个物理机上部署多个es节点，这会导致我们的集群管理更加的复杂。

6、JVM

对于最新的es版本，一般多建议用最新的jvm版本，除非es明确说明要用哪个jdk版本。es和lucene都是一种满足特殊需求的软件，lucene的单元测试和集成测试中，经常会发现jvm自身的一些bug。这些bug涵盖的范围很广，因此尽量用最新的jvm版本，bug会少一些。

就目前es 5.x版本而言，建议用jdk 8，而不是jdk 7，同时jdk 6已经不再被支持了。

如果我们用java编写es应用程序，而且在使用transport client或者node client，要确保运行我们的应用程序的jvm版本跟es服务器运行的jvm版本是一样的。在es中，有些java的本地序列化机制都被使用了，比如ip地址，异常信息，等等。而jvm可能在不同的minor版本之间修改序列化格式，所以如果client和server的jvm版本不一致，可能有序列化的问题。

同时官方推荐，绝对不要随便调整jvm的设置。虽然jvm有几百个配置选项，而且我们可以手动调优jvm的几乎方方面面。同时遇到一个性能场景的时候，每个人都会第一时间想到去调优jvm，但是es官方还是推荐我们不要随便调节jvm参数。因为es是一个非常复杂的分布式软件系统，而且es的默认jvm配置都是基于真实业务场景中长期的实践得到的。随便调节jvm配置反而有可能导致集群性能变得更加差，以及出现一些未知的问题。反而是很多情况下，将自定义的jvm配置全部删除，性能是保持的最好的。

7、容量规划

在规划你的es集群的时候，一般要规划你需要多少台服务器，每台服务器要有多少资源，能够支撑你预计的多大的数据量。但是这个东西其实不是一概而论的，要视具体的读写场景，包括你执行多么复杂的操作，读写QPS来决定的。不过一般而言，根据讲师的实际经验，对于很多的中小型公司，都是建议es集群承载的数据量在10亿规模以内。用最合理的技术做最合理的事情。

这里可以给出几个在国内es非常适合的几个场景，es是做搜索的，当然可以做某个系统的搜索引擎。比如网站或者app的搜索引擎，或者是某些软件系统的搜索引擎，此外es还可以用来做数据分析。那么针对这几个不同的场景，都可以给出具体建议。比如做网站或者app的搜索引擎，一般数据量会相对来说大一些，但是通常而言，一个网站或者app的内容都是有限的，不会无限膨胀，通常数据量从百万级到亿级不等，因此用于搜索的数据都放在es中是合理的。

然后一些软件系统或者特殊项目的搜索引擎，根据项目情况不同，数据量也是从百万量级到几十亿，甚至几百亿，或者每日增量几亿，都有可能，那么此时就要根据具体的业务场景来决定了。如果数据量特别大，日增量都几亿规模，那么其实建议不要将每天全量的数据都写入es中，es也不适合这种无限规模膨胀的场景。es是很耗费内存的，无限膨胀的数据量，会导致我们无法提供足够的资源来支撑这么大的数据量。可以考虑是不是就将部分热数据，比如最近几天的数据，放到es中做高频高性能搜索，然后将大量的很少访问的冷数据放大数据系统做离线批量处理，比如hadoop系统里面。

比如说，你预计一下，你的数据量有多大，需要多少台机器，每台机器要多少资源，来支撑，可以达到多大的性能

数据量 -> 性能，10亿 -> 1s

es达到ms级的化，你必须要有足够的os cache去缓存几乎大部分的索引数据

10亿，每条数据是多大，比如多少个字节，1k -> 100G

5台，64G，8核，300G -> 100G总数据量，300G，一般要分给es jvm heap，150G -> 100G，100G落地到磁盘文件加入很多es自己的信息，100G -> 200G

200G落地磁盘的数据，物理内存剩余的只有150G，可能还有一些操作系统，还有其他的损耗100G

200G落地磁盘的数据，100G物理内存可以用来做os cache，50%的概率是基于os cache做磁盘索引文件的读写，几秒，很正常啦。。。

根据我们的实践经验而言，一般来说，除非是你的机器的内存资源，完全可以容纳所有的落地的磁盘文件的os cache，ms，否则的话，如果不是的话，会大量走磁盘，几秒

同时如果数据量在10亿以内的规模，那么一般而言，如果提供5台以上的机器，每台机器的配置到8核64G的配置，一般而言都能hold住。当然，这个也看具体的使用场景，如果你读写特别频繁，或者查询特别复杂，那么可能还需要更多的机器资源。如果你要承载更大的数据量，那么就相应的提供更多的机器和资源。

要提升你的es的性能，最重要的，还是说规划合理的数据量，物理内存资源大小，os cache

## ES 生产集群部署-从零开始搭建一套 4 个节点 2核4G虚拟机集群

这块我们跟大家说一下，一般来说，你即使要围绕es搭建一个集群的话，也建议至少4~5个节点，因为其实不光是es，后面可能还有kibana，logstash，elk生态栈的其他的东西要部署，包括这个高手进阶篇的课程，后面也是会不断的免费升级的，后面再出几个课程，单独收费，讲解运维，项目，ELK，等等。但是后来觉得说，既然大家都很支持我们，我们说还是决定敢说，对已经购买高级篇的同学，会不断免费升级，第二版，运维，第三版，搜索项目，第四版，ELK

部署一个4个节点的虚拟机集群，每个虚拟机是2核4G，我的笔记本是24G，双核的，16G，宿主机留8G内存

用2核4G，虚拟cpu core，4台，基本可以去模拟真实的生产环境

包括我们接下来的一些部署，都尽量用接近生产环境的标准去部署，让大家可以体验一下

1、在虚拟机中安装CentOS

启动一个virtual box虚拟机管理软件（vmware，我早些年，发现不太稳定，主要是当时搭建一个hadoop大数据的集群，发现每次休眠以后再重启，集群就挂掉了）

virtual box，发现很稳定，集群从来不会随便乱挂，所以就一直用virtual box了

安装virtual box

用的是什么centos镜像，CentOS比较新的版本是7了，然后服务器上装操作系统的话，内存一般比较大，一般是安装64位的，32位的有一个最大内存4G的限制

（1）使用课程提供的CentOS 7镜像即可，CentOS-7-x86\_64-Minimal-1611.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为elasticsearch01，选择操作系统为Linux，选择版本为Red Hat-64bit，分配4096MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。

（4）安装虚拟机中的CentOS 7操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 7镜像文件），按照课程选择后自动安装即可

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

vi /etc/sysconfig/network-scripts/ifcfg-enp0s3

先让它动态分配一个ip地址

ONBOOT=yes

service network restart

ip addr

再设置静态ip地址

BOOTPROTO=static

IPADDR=192.168.31.250

NETMASK=255.255.255.0

GATEWAY=192.168.31.1

service network restart

ip addr

配置DNS

检查NetManager的状态：systemctl status NetworkManager.service

检查NetManager管理的网络接口：nmcli dev status

检查NetManager管理的网络连接：nmcli connection show

设置dns：nmcli con mod enp0s3 ipv4.dns "114.114.114.114 8.8.8.8"

让dns配置生效：nmcli con up enp0s3

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

systemctl stop firewalld.service

systemctl disable firewalld.service

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

yum clean all

yum makecache

yum install wget

------------------------------------------------------------------------------------------

2、在每个CentOS中都安装Java

WinSCP，就是在windows宿主机和linux虚拟机之间互相传递文件的一个工具

（1）安装JDK

1、将jdk-8u131-linux-x64.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-8u131-linux-x64.rpm

3、配置jdk相关的环境变量

vi .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

------------------------------------------------------------------------------------------

3、在4个虚拟机中安装CentOS集群

（1）按照上述步骤，再安装三台一模一样环境的linux机器

（2）另外三台机器的hostname分别设置为elasticsearch02，elasticsearch03，elasticsearch04

（3）安装好之后，在每台机器的hosts文件里面，配置好所有的机器的ip地址到hostname的映射关系

比如说，在elasticsearch01的hosts里面

192.168.31.250 elasticsearch01

192.168.31.xxx elasticsearch02

192.168.31.xxx elasticsearch03

192.168.31.xxx elasticsearch04

------------------------------------------------------------------------------------------

4、配置4台CentOS为ssh免密码互相通信

（1）首先在三台机器上配置对本机的ssh免密码登录

ssh-keygen -t rsa

生成本机的公钥，过程中不断敲回车即可，ssh-keygen命令默认会将公钥放在/root/.ssh目录下

cd /root/.ssh

cp id\_rsa.pub authorized\_keys

将公钥复制为authorized\_keys文件，此时使用ssh连接本机就不需要输入密码了

（2）接着配置三台机器互相之间的ssh免密码登录

使用ssh-copy-id -i hostname命令将本机的公钥拷贝到指定机器的authorized\_keys文件中

## ES 生产集群部署-部署 3 个 ES5.5节点以及 zen discovery 集群发现机制

生产环境集群部署

一点一点讲解的，生产环境去部署的时候，涉及到很多的配置，还有牵扯到了很多的es的知识点

我们先下载es 5.5（7月6号为止）的压缩包，部署到3个节点上面去，但是不启动，因为我们接下来要花费很多讲的时间来讲解各种生产环境的参数的配置

es模拟生产环境的3节点的集群给启动起来，停止es进程，curl，kibana

1、在三个节点上都下载es

如果要安装es，首先就要从官网下载es的安装包，并且最新es版本要求有JDK 8以上的版本

es安装包的目录结构大致如下：

bin：存放es的一些可执行脚本，比如用于启动进程的elasticsearch命令，以及用于安装插件的elasticsearch-plugin插件

conf：用于存放es的配置文件，比如elasticsearch.yml

data：用于存放es的数据文件，就是每个索引的shard的数据文件

logs：用于存放es的日志文件

plugins：用于存放es的插件

script：用于存放一些脚本文件

2、zen discovery集群发现机制

你会在多台机器上，每台机器部署一个es进程，每台机器都启动一个es进程，你怎么让多台机器上的多个es进程，互相发现对方，然后完美的组成一个生产环境的es集群呢？？？。。。。

默认情况下，es进程会绑定在自己的回环地址上，也就是127.0.0.1，然后扫描本机上的9300~9305端口号，尝试跟那些端口上启动的其他es进程进行通信，然后组成一个集群。这对于在本机上搭建es集群的开发环境是很方便的。但是对于生产环境下的集群是不行的，需要将每台es进程绑定在一个非回环的ip地址上，才能跟其他节点进行通信，同时需要使用集群发现机制来跟其他节点上的es node进行通信。

大家还记不记得，我们如果在windows上自己玩儿的话，是不是说，你直接启动多个es进程，他们自己就会组成一个集群

在生产环境中的多台机器上部署es集群，就涉及到了es的discovery机制，也就是集群中各个节点互相发现然后组成一个集群的机制，同时discovery机制也负责es集群的master选举，关于master，一会儿会讲解s

master node和data node两种角色

es是一种peer to peer，也就是p2p点对点的分布式系统架构，不是hadoop生态普遍采用的那种master-slave主从架构的分布式系统。集群中的每个node是直接跟其他节点进行通信的，而不是hadoop生态系统中的那种master-slave分布式系统架构。几乎所有的API操作，比如index，delete，search，等等，都不是说client跟master通信，而是client跟任何一个node进行通信，那个node再将请求转发给对应的node来进行执行。这块的原理，路由，读写原理，在入门篇都讲解过了。

两个角色，master node，data node。正常情况下，就只有一个master node。master node的责任就是负责维护整个集群的状态信息，也就是一些集群元数据信息，同时在node加入集群或者从集群中下限覅按时，重新分配shard，或者是创建或删除了一个索引。包括每次cluster state如果有改变的化，那么master都会负责将集群状态同步给所有的node。

master node负责接收所有的cluster state相关的变化信息，然后将这个改变后的最新的cluster state推动给集群中所有的data node，集群中所有的node都有一份完整的cluster state。只不过master node负责维护而已。其他的node，除了master之外的node，就是负责数据的存储和读写的，写入索引，搜索数据，data node。

如果要让多个node组成一个es集群，首先第一个要设置的参数，就是cluster.name，多个node的cluster.name如果一样，才满足组成一个集群的基本条件。

这个cluster.name的默认值是elasticsearch，在生产环境中，一定要修改这个值，否则可能会导致未知的node无端加入集群，造成集群运行异常。

而es中默认的discovery机制，就是zen discovery机制

zen discovery机制提供了unicast discovery集群发现机制，集群发现时的节点间通信是依赖的transport module，也就是es底层的网络通信模块和协议。

es默认配置为使用unicast集群发现机制，以让经过特殊配置的节点可以组成一个集群，而不是随便哪个节点都可以组成一个集群。但是默认配置下，unicast是本机，也就是localhost，因此只能在一台机器上启动多个node来组成一个集群。虽然es还是会提供multicast plugin作为一个发现机制，但是已经不建议在生产环境中使用了。虽然我们可能想要multicast的简单性，就是所有的node可以再接收到一条multicast ping之后就立即自动加入集群。但是multicast机制有很多的问题，而且很脆弱，比如网络有轻微的调整，就可能导致节点无法发现对方。因此现在建议在生产环境中用unicast机制，提供一个es种子node作为中转路由节点就可以了。

（0）master node、data node、network.host

给集群规划出专门的master eligible node和data node

master node，master eligible node，data node

你配置的时候，是配置多个node变成master eligible node，但是只是说，从这些master eligible node选举一个node出来作为master node，其他master eligible node只是接下来有那个master node故障的时候，接替他的资格，但是还是作为data node去使用的

一般建议master eligible node给3个即可：node.master: true，node.data: false

剩下的node都设置为data node：node.master: false，node.data: true

但是如果一个小集群，就10个以内的节点，那就所有节点都可以作为master eligible node以及data node即可，超过10个node的集群再单独拆分master和data node吧

如果你的节点数量小于10个，小集群，那所有的node，就不要做额外的配置了，master eligible node，同时也是data node

默认情况下，es会将自己绑定到127.0.0.1上，对于运行一个单节点的开发模式下的es是ok的。但是为了让节点间可以互相通信以组成一个集群，需要让节点绑定到一个ip地址上，非会换的地址，一般会配置：network.host: 192.168.1.10。一旦我们配置了network.host，那么es就会认为我们从开发模式迁移到生产模式，同时会启用一系列的bootstrap check。

（1）ping

ping是一个node用discovery机制来发现其他node的一个过程

（2）unicast

unicast discovery集群发现机制是要求配置一个主机列表，用来作为gossip（流言式）通信协议的路由器。这些机器如果通过hostname来指定，那么在ping的时候会被解析为ip地址。unicast discovery机制最重要的两个配置如下所示：

hosts：用逗号分割的主机列表

hosts.resolve\_timeout：hostname被DNS解析为ip地址的timeout等待时长

简单来说，如果要让多个节点发现对方并且组成一个集群，那么就得有一个中间的公共节点，然后不同的节点就发送请求到这些公共节点，接着通过这些公共节点交换各自的信息，进而让所有的node感知到其他的node存在，并且进行通信，最后组成一个集群。这就是基于gossip流言式通信协议的unicast集群发现机制。

当一个node与unicast node list中的一个成员通信之后，就会接收到一份完整的集群状态，这里会列出集群中所有的node。接着那个node再通过cluster state跟master通信，并且加入集群中。这就意味着，我们的unicast list node是不需要列出集群中的所有节点的。只要提供少数几个node，比如3个，让新的node可以连接上即可。如果我们给集群中分配了几个节点作为专门的master节点，那么只要列出我们那三个专门的master节点即可。用如下的配置即可：discovery.zen.ping.unicast.hosts: ["host1", "host2:port"]。

cluster.name

node.name

network.host

discovery.zen.ping.unicast.hosts

（1）已经初步配置好了，各个节点，首先通过network.host绑定到了非回环的ip地址，从而可以跟其他节点通信

（2）通过discovery.zen.ping.unicast.hosts配置了一批unicast中间路由的node

（3）所有node都可以发送ping消息到路由node，再从路由node获取cluster state回来

（4）接着所有node会选举出一个master

（5）所有node都会跟master进行通信，然后加入master的集群

（6）要求cluster.name必须一样，才能组成一个集群

（7）node.name就标识出了每个node我们自己设置的一个名称

（3）master选举

在ping发现过程中，为集群选举出一个master也是很重要的，es集群会自动完成这个操作。这里建议设置discovery.zen.ping\_timeout参数（默认是3s），如果因为网络慢或者拥塞，导致master选举超时，那么可以增加这个参数，确保集群启动的稳定性。

在完成一个集群的master选举之后，每次一个新的node加入集群，都会发送一个join request到master node，可以设置discovery.zen.join\_timeout保证node稳定加入集群，增加join的timeout等待时长，如果一次join不上，默认会重试20次。

如果master node被停止了，或者自己宕机了，那么集群中的node会再次进行一次ping过程，并且选举出一个新的master。如果discovery.zen.master\_election.ignore\_non\_master\_pings设置为了true，那么会强制区分master候选节点，如果node的node.master设置为了false，还来发送ping请求参与master选举，那么这些node会被忽略掉，因为他们没有资格参与。

discovery.zen.minimum\_master\_nodes参数用于设置对于一个新选举的master，要求必须有多少个master候选node去连接那个新选举的master。而且还用于设置一个集群中必须拥有的master候选node。如果这些要求没有被满足，那么master node就会被停止，然后会重新选举一个新的master。这个参数必须设置为我们的master候选node的quorum数量。一般避免说只有两个master候选node，因为2的quorum还是2。如果在那个情况下，任何一个master候选节点宕机了，集群就无法正常运作了。

（4）集群故障的探查

es有两种集群故障探查机制，第一种是通过master进行的，master会ping集群中所有的其他node，确保它们是否是存活着的。第二种，每个node都会去ping master node来确保master node是存活的，否则就会发起一个选举过程。

有下面三个参数用来配置集群故障的探查过程：

ping\_interval：每隔多长时间会ping一次node，默认是1s

ping\_timeout：每次ping的timeout等待时长是多长时间，默认是30s

ping\_retries：如果一个node被ping多少次都失败了，就会认为node故障，默认是3次

（5）集群状态更新

master node是集群中唯一一个可以对cluster state进行更新的node。master node每次会处理一个集群状态的更新事件，应用这次状态更新，然后将更新后的状态发布到集群中所有的node上去。每个node都会接收publish message，ack这个message，但是不会应用这个更新。如果master没有在discovery.zen.commit\_timeout指定的时间内（默认是30s），从至少discovery.zen.minimum\_master\_nodes个节点获取ack响应，那么这次cluster state change事件就会被reject，不会应用。

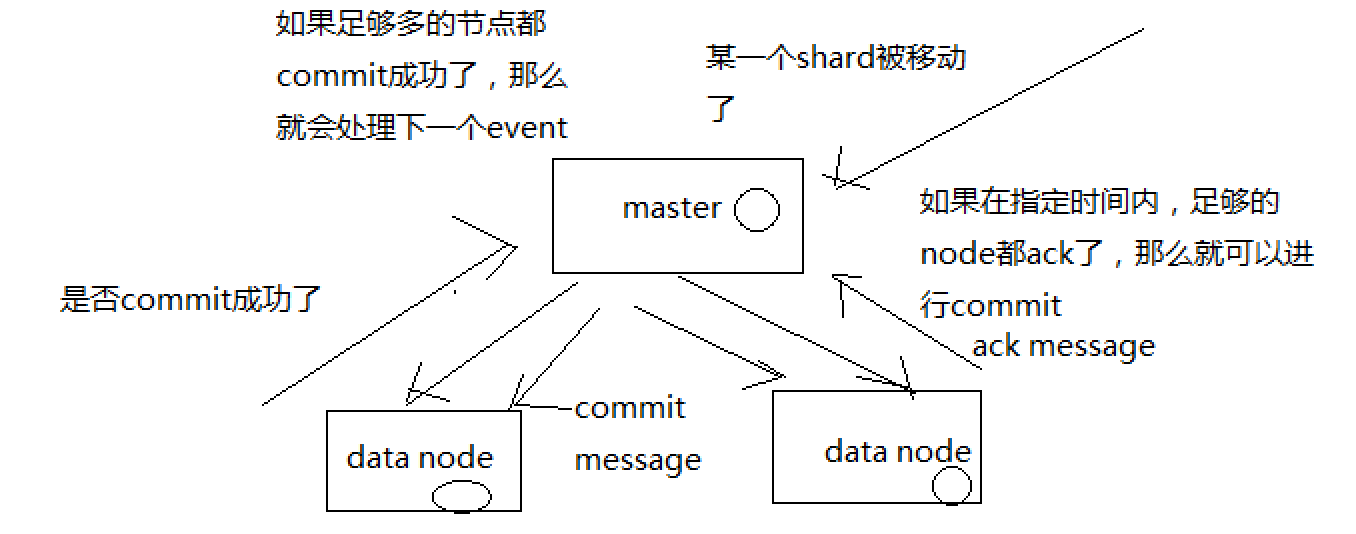
但是一旦在指定时间内，指定数量的node都返回了ack消息，那么cluster state就会被commit，然后一个message会被发送给所有的node。所有的node接收到那个commit message之后，接着才会将之前接收到的集群状态应用到自己本地的状态副本中去。接着master会等待所有节点再次响应是否更新自己本地副本状态成功，在一个等待超时时长内，如果接收到了响应，那么就会继续处理内存queue中保存的下一个更新状态。discovery.zen.publish\_timeout默认是30s，这个超时等待时长是从plublish cluster state开始计算的。

（6）不因为master宕机阻塞集群操作

如果要让集群正常运转，那么必须有一个master，还有discovery.zen.minimum\_master\_nodes指定数量的master候选node，都在运行。discovery.zen.no\_master\_block可以控制当master当即时，什么样的操作应该被拒绝。有下面两个选项：

all：一旦master当即，那么所有的操作都会被拒绝

write：这是默认的选项，所有的写操作都会被拒绝，但是读操作是被允许的



## ES 生产集群部署-根据自己的集群设置一些重要参数

1、es的默认参数

es的默认参数是非常好的，适合绝大多数的情况，尤其是一些性能相关的配置。因此刚开始部署一个生产环境下的es集群时，几乎所有的配置参数都可以用默认的设置。有很多的生产环境场景下，都是因为es集群管理人员自己去调整es的某些配置，结果导致集群出现了严重的故障，那些es集群管理员甚至还以为做出那些调节可以将es性能提升一百倍以上。

比如mysql或者oracle这种关系型数据库，也许是需要非常重的调优，但是es是真的不用。如果我们现在面临着一些es的性能问题，通常建议的解决方案是更好的进行数据结构的布局，或者增加更多的节点和机器资源。在es的性能调优中，真的很少有那种magic knobs，就是某个参数一调节，直接性能提升上百倍。即使有这种参数，es官方也早就将其设置为默认的最佳值了。

但是在生产环境中，还是有极少数跟公司和业务相关的配置是需要我们修改的。这些设置都是具体的公司和业务相关联的，是没法预先给予最好的默认配置的。

2、集群名称和节点名称

默认情况下，es会启动一个名称为elasticsearch的集群。通常建议一定要将自己的集群名称重新进行命名，主要是避免公司网络环境中，也许某个开发人员的开发机会无意中加入你的集群。比如说将你的集群名称命名为elasticsearch\_production。在elasticsearch.yml中，可以设置集群名称：cluster.name: elasticsearch\_production。

此外，每个node启动的时候，es也会分配一个随机的名称。这个也不适合在生产环境中，因为这会导致我们没法记住每台机器。而且每次重启节点都会随机分配，就导致node名称每次重启都会变化。因此通常我们在生产环境中是需要给每个node都分配一个名称的。在elasticsearch.yml中配置即可：node.name: elasticsearch\_005\_data。

3、文件路径

（1）数据目录、日志目录以及插件目录

默认情况下，es会将plugin，log，还有data ，config，file都放在es的安装目录中。这有一个问题，就是在进行es升级的时候，可能会导致这些目录被覆盖掉。导致我们丢失之前安装好的plugin，已有的log，还有已有的数据，以及配置好的配置文件。

所以一般建议在生产环境中，必须将这些重要的文件路径，都重新设置一下，放在es安装目录之外。path.data用于设置数据文件的目录，path.logs用于设置日志文件的目录，path.plugins用于设置插件存放的目录。path.data可以指定多个目录，用逗号分隔即可。如果多个目录在不同的磁盘上，那么这就是一个最简单的RAID 0的方式，将数据在本地进行条带化存储了，可以提升整体的磁盘读写性能。es会自动将数据在多个磁盘的多个目录中条带化存储数据。

一般建议的目录地址是：

mkdir -p /var/log/elasticsearch

mkdir -p /var/data/elasticsearch

mkdir -p /var/plugin/elasticsearch

mkdir -p /etc/elasticsearch

path.logs: /var/log/elasticsearch

path.data: /var/data/elasticsearch

path.plugins: /var/plugin/elasticsearch

config：/etc/elasticsearch

在RAID 0的存储级别下，每个磁盘上回存储一部分数据，但是如果一个磁盘故障了，那么可能导致这台机器上的部分数据就丢失了。如果我们的es是有replica的，那么在其他机器上还是会有一份副本的。如果data file指定了多个目录，为了尽量减少数据丢失的风险，es会将某个shard的数据都分配到一个磁盘上去。这就意味着每个shard都仅仅会放在一个磁盘上。es不会将一个shard的数据条带化存储到多个磁盘上去，因为如果一个磁盘丢失了，就会导致整个shard数据丢失。

但是这又引入了性能的问题，如果我们给一个机器添加更多的磁盘来提升单个索引的读写性能，是没有效果的。因为这个索引在这个机器上的shard仅仅存在于一个磁盘上。因此data file指定多个目录，仅仅对于你的一台机器上存储了多个index的多个shard时，才会有效果的。因为不同index的shard可能就被存储到不同的磁盘上去了，对多个index的shard读写可以走不同磁盘，提升了性能。

虽然multiple data path是一个很有用的功能，但是es毕竟不是一个专门的RAID软件。如果我们要对RAID存储策略进行更多的配置，提高存储的健壮性以及灵活性，还是要用专门的RAID软件来进行机器的磁盘数据存储，而不是用multiple data path策略。

综上所述，multiple data path功能在实际的生产环境中，其实是较少使用的。

（2）配置文件目录

es有两个配置文件，elasticsearch.yml，用于配置es，还有一个log4j.properties用来配置es日志打印。这些文件都被放在config目录下，默认就是ES\_HOME/config。可以通过下面的命令来重新设置：./bin/elasticsearch -Epath.conf=/path/to/my/config/。

配置文件的格式是yaml格式的，比如下面这种格式：

path:

data: /var/lib/elasticsearch

logs: /var/log/elasticsearch

path.data: /var/lib/elasticsearch

path.logs: /var/log/elasticsearch

4、日志配置

es使用log4j2来记录日志，log4j2可以通过log4j2.properties文件来进行配置。比如下面的这份配置文件：

appender.rolling.type = RollingFile

appender.rolling.name = rolling

appender.rolling.fileName = ${sys:es.logs.base\_path}${sys:file.separator}${sys:es.logs.cluster\_name}.log

appender.rolling.layout.type = PatternLayout

appender.rolling.layout.pattern = [%d{ISO8601}][%-5p][%-25c] %.10000m%n

appender.rolling.filePattern = ${sys:es.logs.base\_path}${sys:file.separator}${sys:es.logs.cluster\_name}-%d{yyyy-MM-dd}.log

appender.rolling.policies.type = Policies

appender.rolling.policies.time.type = TimeBasedTriggeringPolicy

appender.rolling.policies.time.interval = 1

appender.rolling.policies.time.modulate = true

appender.rolling.type = RollingFile，就配置了appender类型是RollingFile

appender.rolling.fileName = ${sys:es.logs.base\_path}${sys:file.separator}${sys:es.logs.cluster\_name}.log，就配置了日志路径是/var/log/elasticsearch/production.log

appender.rolling.filePattern = ${sys:es.logs.base\_path}${sys:file.separator}${sys:es.logs.cluster\_name}-%d{yyyy-MM-dd}.log，就配置了将日志每天写一份到/var/log/elasticsearch/production-2017-01-01.log文件中

appender.rolling.policies.time.type = TimeBasedTriggeringPolic，这里配置了用基于时间的roll策略

appender.rolling.policies.time.interval = 1，这个设置了每天一份日志文件

appender.rolling.policies.time.modulate = true，这个设置了根据自然天来划分文件，而不是24小时

还可以配置将日志文件保留一段时间内，同时删除之前的日志文件

appender.rolling.strategy.type = DefaultRolloverStrategy

appender.rolling.strategy.action.type = Delete

appender.rolling.strategy.action.basepath = ${sys:es.logs.base\_path}

appender.rolling.strategy.action.condition.type = IfLastModified

appender.rolling.strategy.action.condition.age = 7D

appender.rolling.strategy.action.PathConditions.type = IfFileName

appender.rolling.strategy.action.PathConditions.glob = ${sys:es.logs.cluster\_name}-\*

第一行是配置了默认的DefaultRolloverStrategy

第二行是配置了Delete action，在rollover之后，就会删除文件

第三行是配置了es log的基础路径

第四行是配置了rollover发生的条件，是基于IfLastModified

第五行是配置了保留的天数，这里是7天

第六行是配置了删除匹配7天前的文件

第七行是配置了一个删除文件的格式，这样就只是删除过期日志文件，但是不要删除慢查询日志

## ES 生产集群部署-针对生产集群的脑裂问题专门定制的重要参数

最少master候选节点以及脑裂问题

discovery.zen.minimum\_master\_nodes参数对于集群的可靠性来说，是非常重要的。这个设置可以预防脑裂问题，也就是一个集群中存在两个master。

如果因为网络的故障，导致一个集群被划分成了两片，每片都有多个node，以及一个master，那么集群中就出现了两个master了。但是因为master是集群中非常重要的一个角色，主宰了集群状态的维护，以及shard的分配，因此如果有两个master的化，可能会导致破坏数据。

那么那个参数的作用，就是告诉es直到有足够的master候选节点时，才可以选举出一个master，否则就不要选举出一个master。这个参数必须被设置为集群中master候选节点的quorum数量，也就是大多数。至于quorum的算法，就是：master候选节点数量 / 2 + 1。

比如我们有10个节点，都能维护数据，也可以是master候选节点，那么quorum就是10 / 2 + 1 = 6。

如果我们有三个master候选节点，还有100个数据节点，那么quorum就是3 / 2 + 1 = 2

如果我们有2个节点，都可以是master候选节点，那么quorum是2 / 2 + 1 = 2。此时就有问题了，因为如果一个node挂掉了，那么剩下一个master候选节点，是无法满足quorum数量的，也就无法选举出新的master，集群就彻底挂掉了。此时就只能将这个参数设置为1，但是这就无法阻止脑裂的发生了。

2个节点，discovery.zen.minimum\_master\_nodes分别设置成2和1会怎么样

综上所述，一个生产环境的es集群，至少要有3个节点，同时将这个参数设置为quorum，也就是2。discovery.zen.minimum\_master\_nodes设置为2，如何避免脑裂呢？

那么这个是参数是如何避免脑裂问题的产生的呢？比如我们有3个节点，quorum是2.现在网络故障，1个节点在一个网络区域，另外2个节点在另外一个网络区域，不同的网络区域内无法通信。这个时候有两种情况情况：

（1）如果master是单独的那个节点，另外2个节点是master候选节点，那么此时那个单独的master节点因为没有指定数量的候选master node在自己当前所在的集群内，因此就会取消当前master的角色，尝试重新选举，但是无法选举成功。然后另外一个网络区域内的node因为无法连接到master，就会发起重新选举，因为有两个master候选节点，满足了quorum，因此可以成功选举出一个master。此时集群中就会还是只有一个master。

（2）如果master和另外一个node在一个网络区域内，然后一个node单独在一个网络区域内。那么此时那个单独的node因为连接不上master，会尝试发起选举，但是因为master候选节点数量不到quorum，因此无法选举出master。而另外一个网络区域内，原先的那个master还会继续工作。这也可以保证集群内只有一个master节点。

综上所述，通过在elasticsearch.yml中配置discovery.zen.minimum\_master\_nodes: 2，就可以避免脑裂问题的产生。

但是因为es集群是可以动态增加和下线节点的，所以可能随时会改变quorum。所以这个参数也是可以通过api随时修改的，特别是在节点上线和下线的时候，都需要作出对应的修改。而且一旦修改过后，这个配置就会持久化保存下来。

PUT /\_cluster/settings

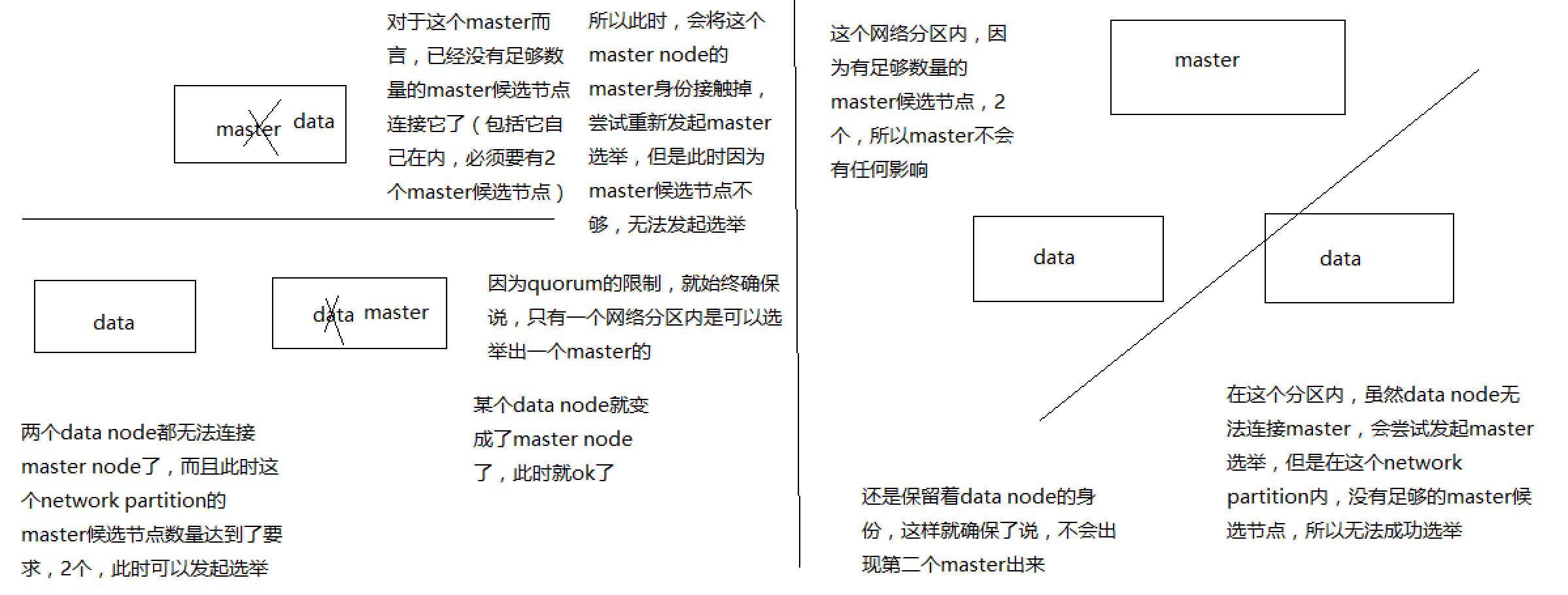
{

"persistent" : {

"discovery.zen.minimum\_master\_nodes" : 2

}

}



## ES 生产集群部署-集群重启时的 shard 恢复耗时过长问题定制的重要参数

shard recovery配置以及集群重启时的无意义shard重分配问题

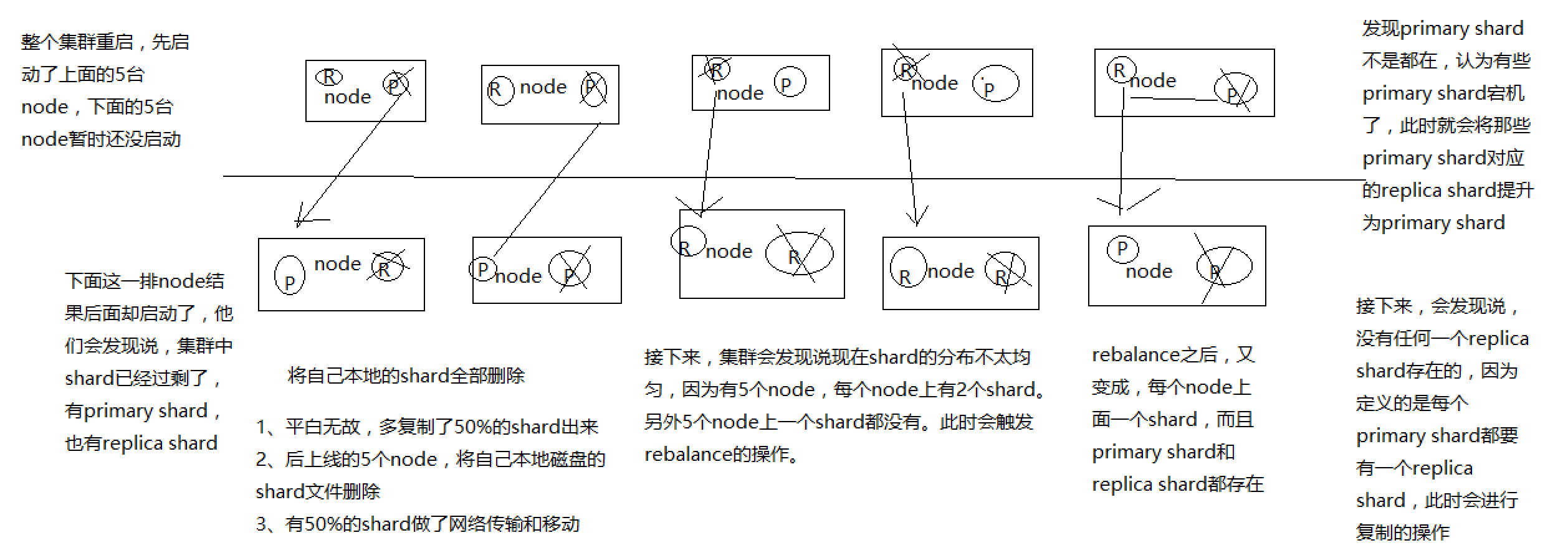
在集群重启的时候，有一些配置会影响shard恢复的过程。首先，我们需要理解默认配置下，shard恢复过程会发生什么事情。如果我们有10个node，每个node都有一个shard，可能是primary shard或者replica shard，你有一个index，有5个primary shard，每个primary shard有一个replica shard。如果我们将整个集群关闭了进行一些维护性的操作，比如给机器安装新的磁盘之类的事情。当我们重启集群的时候，肯定节点是一个接一个的启动的，可能会出现5个节点先启动了，然后剩下5个节点还没启动。

也许是因为剩下的5个节点没来得及启动，或者是因为一些原因耽搁了，总之不管是什么原因，就是现在只有5个节点是在线的。这5个节点会通过gossip协议互相通信，选举出一个master，然后组成一个集群。他们会发现数据没有被均匀的分布，因为有5个节点没有启动，那么那5个节点上的shard就是不可用的，集群中就少了一半的shard。此时在线的5个node就会将部分replica shard提升为primary shard，同时为每个primary shard复制足够的replica shard。

最后，可能剩下的5个节点加入了集群。但是这些节点发现本来是他们持有的shard已经被重新复制并且放在之前的5个node速度回当了，此时他们就会删除自己本地的数据。然后集群又会开始进行shard的rebalance操作，将最早启动的5个node上的shard均匀分布到后来启动的5个node上去。

在这个过程中，这些shard重新复制，移动，删除，再次移动的过程，会大量的耗费网络和磁盘资源。对于数据量庞大的集群来说，可能导致每次集群重启时，都有TB级别的数据无端移动，可能导致集群启动会耗费很长时间。但是如果所有的节点都可以等待整个集群中的所有节点都完全上线之后，所有的数据都有了以后，再决定是否要复制和移动shard，情况就会好很多。

所以现在问题我们已经知道了，那么我们就可以配置一些设置来解决这个问题。首先我们需要设置一个参数，gateway.recover\_after\_nodes: 8。这个参数可以让es直到有足够的node都上线之后，再开始shard recovery的过程。所以这个参数是跟具体的集群相关的，要根据我们的集群中节点的数量来决定。此外，还应该设置一个集群中至少要有多少个node，等待那些node的时间：gateway.expected\_nodes: 10，gateway.recover\_after\_time: 5m。经过上面的配置之后，es集群的行为会变成下面这样，等待至少8个节点在线，然后等待最多5分钟，或者10个节点都在线，开始shard recovery的过程。这样就可以避免少数node启动时，就立即开始shard recovery，消耗大量的网络和磁盘资源，甚至可以将shard recovery过程从数小时缩短为数分钟。



## ES 生产集群部署-绝对不能随意调节 jvm 和 thread pool的原因

es中有很多的配置都让大家忍不住去调优，因为也许大家都太过于迷恋性能优化了，都认为优化一些配置可以大幅度提升性能，就感觉性能调优像个魔法一样，是个万能的东西。但是其实99.99%的情况下，对于es来说，大部分的参数都保留为默认的就可以了。因为这些参数经常被滥用和错误的调节，继而导致严重的稳定性问题以及性能的急剧下降。

1、jvm gc

jvm使用垃圾回收器来释放掉不用的内存，千万不要去调节默认的垃圾回收行为。es默认用的垃圾回收器是CMS。CMS回收器是并发式的回收器，能够跟应用程序工作线程并发工作，最大程度减少垃圾回收时的服务停顿时间。但是CMS还是会有两个停顿阶段，同时在回收特别大的heap时也会有一些问题。尽管有一些缺点，但是CMS对于要求低延时请求响应的软件来说，还是最佳的垃圾回收器，因此官方的推荐就是使用CMS垃圾回收器。

有一种最新的垃圾回收器叫做G1。G1回收器可以比CMS提供更少的回收停顿时间，而且能够这对大heap有更好的回收表现。它会将heap划分为多个region，然后自动预测哪个region会有最多可以回收的空间。通过回收那些region，就可以最小化停顿时长，而且可以针对大heap进行回收。

听起来还挺不错的，但是不幸的是，G1还是比较年轻的一种垃圾回收器，而且经常会发现一些新的bug，这些bug可能会导致jvm挂掉。lucene的测试套件就检查出来了G1的一些bug。因此es官方不推荐现在使用G1垃圾回收器，也许在不久的未来，等G1更加稳定的时候，可以使用G1。

2、threadpool

每个人都很喜欢去调优线程池，而且大部分人都特别喜欢增加线程池的线程数量，无论是大量的写入，还是大量的搜索，或者是感觉服务器的cpu idle空闲率太高，都会增加更多的线程。在es中，默认的threadpool设置是非常合理的，对于所有的threadpool来说，除了搜索的线程池，都是线程数量设置的跟cpu core一样多的。如果我们有8个cpu core，那么就可以并行运行8个线程。那么对于大部分的线程池来说，分配8个线程就是最合理的数量。

不过搜索会有一个更加大的threadpool，一般被配置为：cpu core \* 3 / 2 + 1。

也许我们会觉得有些线程可能会因为磁盘IO等操作block住，所以我们需要更多的线程。但是在es中这并不是一个问题，大多数的磁盘IO操作都是由lucene的线程管理的，而不是由es管理的，因此es的线程不需要关心这个问题。此外，threadpool还会通过在彼此之间传递任务来协作执行，我们不需要担心某一个网络线程会因为等待一次磁盘写操作，而导致自己被block住，无法处理网络请求。网络线程可以将那个磁盘写操作交给其他线程池去执行，然后自己接着回来处理网络请求。

其实我们的进程的计算能力是有限的，分配更多的线程只会强迫cpu在多个线程上下文之间频繁来回切换。一个cpu core在同一时间只能运行一条线程，所以如果cpu要切换到另外一个线程去执行，需要将当前的state保存起来，然后加载其他的线程进来执行。如果线程上下文切换发生在一个cpu core内，那么还好一些，但是如果在多个cpu core之间发生线程上下文切换，那么还需要走一个cpu core内部的通信。这种线程上下文切换会消耗掉很多的cpu资源，对于现在的cpu来说，每次线程上下文切换，都会导致30微秒的时间开销，所以宁愿将这些时间花费在任务的处理上。

很多人会将threadpool大小设置为一些很愚蠢的数值，在一个8核的机器上，可能运行了超过60，100，甚至1000个线程。这么多的线程会导致cpu资源利用率很低。所以下次如果我们要调节线程池的话，记住，千万别这么干。如果一定要调节线程数量，也得记住要根据你的cpu core数量来调节，比如设置为cpu core的两倍，如果设置的再多，那么就是一种浪费了。

## ES 生产集群部署-jvm 和服务器内存分配的最佳实践以及原理分析

除了之前讲解的一些配置，根据你的集群环境特殊的配置，我们这一讲来讲解最重要的内存的分配，提出一些问题，生产环境部署es，不可避免要回答一个问题，比如我的机器上有64G的内存，或者32G的内存，那么一般来说我应该分配多少个G的内存给es的jvm heap

1、jvm heap分配

es默认会给jvm heap分配2个G的大小，对于几乎所有的生产环境来说，这个内存都太小了。如果用这个默认的heap size，那么生产环境的集群肯定表现不会太好。

有两个方式来调节es中的jvm heap size。最简单的就是设置环境变量，ES\_HEAP\_SIZE。当es进程启动的时候，会读取这个环境变量的值，然后设置为jvm的heap size。举例来说，可以这样来设置：export ES\_HEAP\_SIZE=10g。此外，还可以在启动es进程的时候，传递一个jvm的option，比如：ES\_JAVA\_OPTS="-Xms10g -Xmx10g" ./bin/elasticsearch，但是要注意-Xms和-Xmx最小和最大堆内存一定设置的一样，避免运行过程中的jvm heap resize，那会是一个非常耗时的过程。

在老版本的es中，比如es 2.x里面，一般推荐用ES\_HEAP\_SIZE环境变量的方式来设置jvm heap size。

在新版本的es中，比如es 5.x里面，一般推荐在jvm.options文件里面去设置jvm相关的参数。

2、将机器上少于一半的内存分配给es

一个常见的问题就是将es进程的jvm heap size设置的过于大了。比如我们有一台64G的机器，可能我们甚至想要给es jvm size设置64G内存。但是这是错误的。大家可能会觉得说，直接将机器上的可用的内存都分配给es jvm heap，性能是绝对高的，因为大量的数据都可以缓存在内存里面。

虽然heap对于es来说是非常重要的，jvm heap被es用来存放很多内存中的数据结构来提供更快的操作性能。但是还有另外一个内存的用户，那就是lucene。lucene的设计就是要使用底层的os filesystem cache来缓存数据结构。lucene的segment是保存在单独的文件中的。因为这些segment是不可变的，所以这些文件实际上也从来不会改变。这样的话，就可以更好的缓存这些文件，底层的os cache会将hot segment驻留在内存中以供更快的访问。这些segment包括了倒排索引（为了全文检索）以及正排索引（为了聚合操作）。lucene的性能是严重依赖于底层的os的，但是如果我们给了过多的内存到es的jvm heap，那么就没有足够的内存留给lucene。这会极大的影响性能。

这里想告诉大家的是，就是说，es的性能很大的一块，其实是由有多少内存留给操作系统的os cache，供lucene去缓存索引文件，来决定的。所以说lucene的os cache有多少是非常重要的。

一般建议的是，将50%的内存分配给es jvm heap，然后留50%的内存给os cache。留给os cache的内存是不会不使用的，lucene会将剩下的内存全部用光，用来cache segment file。如果我们没有对任何分词的text field进行聚合操作，那么我们就不需要使用fielddata，我们甚至可以考虑给os cache更多的内存，因为fielddata是要用jvm heap。如果我们给jvm heap更少的内存，那么实际上es的性能反而会更好，因为更多的内存留给了lucene用os cache提升索引读写性能，同时es的jvm heap的gc耗时会更少。

es部署的机器上，内存是如何分配的，如何使用的，如何决定我们的操作系统的，我们该如何给jvm和os cache分配内存

3、不要给jvm分配超过32G内存

还有另外一个原因不要将过多的内存分配给es的jvm heap。如果heap小于32G的化，jvm会用一种技术来压缩对象的指针，object pointer。在java中，所有的对象都会被分配到heap中，然后被一个pointer给引用。object pointer会指向heap中的对象，引用的是二进制格式的地址。

对于32位的系统来说，jvm最大的heap size就是4G，解释一下，32位，0和1值，0和1在32位的组合是2^32次方的字节，除以1024就是多少k，再除以1024就是多少mb，再除以1024就是多少gb，最后算下来就是4G。对于64位的系统来说，heap size可以更大，但是64位的object pointer会耗费更多的空间，因为object pointer更大了。比浪费更多内存空间更恶劣的是，过大的object pointer会在cpu，main memory和LLC、L1等多级缓存间移动数据的时候，吃掉更多的带宽。

所以jvm用了一种技术，叫做compressed oops来解决object pointer耗费过大空间的问题。这个技术的核心思想是，不要让object pointer引用内存中的二进制地址，而是让object pointer引用object offset。这就意味着32位的pointer可以引用400万个对象，而不是400万字节。这也意味着，使用32位的pointer，最大的heap大小可以到32G。此时只要heap size在32G以内，jvm就会自动启用32位的object pointer，因为32位的对象指针，足够引用32G的内存了，就可以用32位的pointer替代64位的pointer。但是32位的pointer比64位的pointer可以耗费更少的内存耗费。

如果你给jvm heap分配的内存小于32G，此时jvm会自动使用32位的object pointer，同时是让pointer指向对象的offset，32位的object pointer就足以引用32G的内存，同时32位的pointer占用的内存空间很少，对cpu和memory之间移动数据的带宽开销也很少。这个过程就叫做compressed oops。

但是一旦我们越过了32G这个界限，就是给jvm heap分配了超过32G的内存，比较坑了。就没有办法用32位的pointer+引用object offset的模式了，因为32位的pointer最多引用32G的内存，超过了32G，就没法用32位pointer。不用32位pointer，就只能用64位pointer，才能引用超过32G的内存空间。此时pointer就会退回到传统的object pointer引用对象的二进制地址的模式，此时object pinter的大小会急剧增长，更多的cpu到内存的带宽会被占据，更多的内存被耗费。实际上，不用compressed oops时，你如果给jvm heap分配了一个40~50G的内存的可用空间，实际上被object pointer可能都要占据十几G的内存空间，可用的空间量，可能跟使用了compressed oops时的32GB内存的可用空间，20多个G，几乎是一样的。

因此，即使我们有很多内存，但是还是要分配给heap在32GB以内，否则的话浪费更多的内存，降低cpu性能，而且会让jvm回收更大的heap。

综上所述，如果你给jvm heap分配超过32G的内存，实际上是没有什么意义的，因为用64位的pointer，1/3的内存都给object pointer给占据了，这段内存空间就浪费掉了。还不如分配32G以内，启用compressed oops，可用空间跟你分配50个G的内存，是一样的。

所以也正是因为32G的限制，一般来说，都是建议说，如果你的es要处理的数据量上亿的话，几亿，或者十亿以内的规模的话，建议，就是用64G的内存的机器比较合适，有个5台，差不多也够了。给jvm heap分配32G，留下32G给os cache。

4、在32G以内的话具体应该设置heap为多大？

这个是根据具体情况而定的，不是固定死的，根据不同的jvm和平台而变。一般而言，将jvm heap size设置为31G比较安全一些。主要是要确保说，你设置的这个jvm heap大小，可以让es启用compressed oops这种优化机制。此外，可以给jvm option加入-XX:+PrintFlagsFinal，然后可以打印出来UseCompressedOops是否为true。这就可以让我们找到最佳的内存设置。因为可以不断调节内存大小，然后观察是否启用compressed oops。

举例来说，如果在mac os上启动一个java 1.7，同时将heap size设置为32600mb，那么compressed oops是会开启的；但是如果设置为32766m，compressed oops就不会开启。相反的是，使用jdk 1.8的化，分配32766m，compressed oops是会开启的，设置为32767m，就不会开启。所以说，这个东西不是固定的。根据不同的操作系统以及jvm版本而定。

在es启动日志中，我们可以查看compressed oops是否开启，比如下面的字样：[2015-12-16 13:53:33,417][INFO ][env] [Illyana Rasputin] heap size [989.8mb], compressed ordinary object pointers [true]。

5、对于有1TB内存的超大内存机器该如何分配？

如果我们的机器是一台超级服务器，内存资源甚至达到了1TB，或者512G，128G，该怎么办？首先es官方是建议避免用这种超级服务器来部署es集群的，但是如果我们只有这种机器可以用的话，我们要考虑以下几点：

（1）我们是否在做大量的全文检索？考虑一下分配4~32G的内存给es进程，同时给lucene留下其余所有的内存用来做os filesystem cache。所有的剩余的内存都会用来cache segment file，而且可以提供非常高性能的搜索，几乎所有的数据都是可以在内存中缓存的，es集群的性能会非常高

（2）是否在做大量的排序或者聚合操作？聚合操作是不是针对数字、日期或者未分词的string？如果是的化，那么还是给es 4~32G的内存即可，其他的留给es filesystem cache，可以将聚合好用的正排索引，doc values放在os cache中

（3）如果在针对分词的string做大量的排序或聚合操作？如果是的化，那么就需要使用fielddata，这就得给jvm heap分配更大的内存空间。此时不建议运行一个节点在机器上，而是运行多个节点在一台机器上，那么如果我们的服务器有128G的内存，可以运行两个es节点，然后每个节点分配32G的内存，剩下64G留给os cache。如果在一台机器上运行多个es node，建议设置：cluster.routing.allocation.same\_shard.host: true。这会避免在同一台物理机上分配一个primary shard和它的replica shard。

6、swapping

如果频繁的将es进程的内存swap到磁盘上，绝对会是一个服务器的性能杀手。想象一下，内存中的操作都是要求快速完成的，如果需要将内存页的数据从磁盘swap回main memory的化，性能会有多差。如果内存被swap到了磁盘，那么100微秒的操作会瞬间变成10毫秒，那么如果是大量的这种内存操作呢？这会导致性能急剧下降。

因此通常建议彻底关闭机器上的swap，swapoff -a，如果要永久性关闭，需要在/etc/fstab中配置

如果没法完全关闭swap，那么可以尝试调低swappiness至，这个值是控制os会如何将内存swap到磁盘的。这会在正常情况下阻止swap，但是在紧急情况下，还是会swap。一般用sysctl来设置，vm.swappiness = 1。如果swappiness也不能设置，那么就需要启用mlockall，这就可以让我们的jvm lock住自己的内存不被swap到磁盘上去，在elasticsearch.yml中可以设置：bootstrap.mlockall: true。

## ES 生产集群部署-重要的操作系统设置(swapping,virutal memory)

1、系统的重要配置

理想情况下，es应该单独在一个服务器上运行，能够使用服务器上的所有资源。为了达到上述目标，我们需要配置操作系统，来允许用户运行es并且获取比默认情况下更多的资源。

在生产环境中下面的一些设置必须配置一下：

（1）禁止swapping

（2）确保拥有足够的虚拟内存

（3）确保拥有足够的线程数量

开发模式 vs 生产模式

默认情况下，es会假设你是在开发模式下运行的。如果上面的任何配置没有正确的设置，那么会输出一些warning到日志文件中，但是我们还是可以启动es进程的。

但是如果我们配置了网络设置，比如network.host，es会认为我们是运行在生产环境中的，然后就会将上述warning升级为exception。这些exception会阻止我们的es节点启动。这是一个重要的安全保障措施来确保我们不会因为错误的配置了es server，而导致数据丢失。

2、配置系统设置

在/etc/security/limits.conf中，可以配置系统设置

也可以用ulimit临时配置系统设置

在linux操作系统中，ulimit可以用来临时的改变资源限制。通常需要用root权限来设置ulimit。

举例，如果要设置file descriptor为65536，可以用如下的命令来设置：

ulimit -n 65536

但是在linux操作系统中，实际上永久性的资源限制可以通过编辑/etc/security/limits.conf文件来设置。比如要设置file descriptor，可以再limits.conf中加入下面的行：

elasticsearch - nofile 65536

在下一次elasticsearch用户开启一个新的会话时就会生效

设置jvm option

一般建议通过jvm.options配置文件来设置es的jvm option。默认的地址是config/jvm.options

每行是一个jvm argument

此外，如也可以通过ES\_JAVA\_OPTS环境变量来设置jvm option，比如下面的命令：

export ES\_JAVA\_OPTS="$ES\_JAVA\_OPTS -Djava.io.tmpdir=/path/to/temp/dir"

3、禁止swapping

大多数操作系统都会使用尽量多的内存来进行file system cache，并且尽量将不经常使用的java应用的内存swap到磁盘中去。这会导致jvm heap的部分内存，甚至是用来执行代码的内存页被swap到磁盘中去。

swapping对于性能来说是非常差劲的，为了es节点的稳定性考虑，应该尽量避免这种swapping。因为swapping会导致gc过程从毫秒级变成分钟级，在gc的时候需要将内存从磁盘中swapping到内存里，特别耗时，这会导致es节点响应请求变得很慢，甚至导致es node跟cluster失联。在一个弹性的分布式系统中，让操作系统kill掉某一个节点，是很高效的。

有三种方法可以disable swapping。推荐的option是彻底禁用swap，如果做不到的化，也得尽量最小化swappiness的影响，比如通过lock memory的方法。

（1）禁用所有的swapping file

通常来说，es进程会在一个节点上单独运行，那么es进程的内存使用是由jvm option控制的。

可以使用下面的命令临时性禁止swap：swapoff -a

要永久性的禁止swap，需要修改/etc/fstab文件，然后将所有包含swap的行都注释掉

（2）配置swappiness

另外一个方法就是通过sysctl，将vm.swappiness设置为1，这可以尽量减少linux内核swap的倾向，在正常情况下，就不会进行swap，但是在紧急情况下，还是会进行swap操作。sysctl -w vm.swappiness=1

（3）启用bootstrap.memory\_lock

最后一个选项，就是用mlockall，将es jvm进程的address space锁定在内存中，阻止es内存被swap out到磁盘上去。在config/elasticsearch.yml中，可以配置：

bootstrap.memory\_lock: true

GET \_nodes?filter\_path=\*\*.mlockall，通过这行命令可以检查mlockall是否开启了

如果发现mlockall是false，那么意味着mlockall请求失败了。会看到一行日志，unable to lock jvm memory。

最大可能的原因，就是在linux系统中，启动es进程的用户没有权限去lock memory，需要通过以下方式进行授权：

ulimit -l unlimited

/etc/security/limits.conf，memlock设置为unlimited

另外一个原因可能是临时目录使用noexec option来mount了。可以通过指定一个新的临时目录来解决

export ES\_JAVA\_OPTS="$ES\_JAVA\_OPTS -Djava.io.tmpdir=/path/to/temp/dir"

当然也可以通过在jvm.options文件中来设置java.io.tmpdir

4、虚拟内存

es使用hybrid mmapfs / niofs目录来存储index数据，操作系统的默认mmap count限制是很低的，可能会导致内存耗尽的异常。

需要提升mmap count的限制：sysctl -w vm.max\_map\_count=262144

如果要永久性设置这个值，要修改/etc/sysctl.conf，将vm.max\_map\_count的值修改一下，重启过后，用sysctl vm.max\_map\_count来验证一下数值是否修改成功

es同时会用NioFS和MMapFS来处理不同的文件，我们需要设置最大的map刷另，这样我们才能有足够的虚拟内存来给mmapped文件使用，可以用sysctl来设置：sysctl -w vm.max\_map\_count=262144。还可以再/etc/sysctl.conf中，对vm.max\_map\_count来设置。

5、设置线程的数量

es用了很多线程池来应对不同类型的操作，在需要的时候创建新的线程是很重要的。要确保es用户能创建的最大线程数量至少在2048以上。

可以通过ulimit -u 2048来临时设置，也可以在/etc/security/limits.conf中设置nproc为2048来永久性设置。

## ES 生产集群部署-production mode 下启动时的 bootstrap check

1、bootstrap check

经常会碰到一些es的用户，遇到一些奇怪的问题，主要是因为他们没有配置一些重要的设置。在es以前的老版本中，对这些设置错误的配置，会在日志里记录一些warning告警。但是有时候用户会忽略这些日志中的告警信息。为了确保说这些设置的错误配置告警信息可以引起用户的注意，es的新版本中引入了bootstrap check，也就是启动时检查。

这些启动时检查操作，会检查许多es和系统的设置，将这些配置的值跟es期望的安全值去进行比较。如果es在development mode下，那么失败的检查仅仅在日志中打印warning。如果es运行在生产模式下，任何启动时检查的失败都会导致es拒绝启动。

2、development mode vs. production mode

默认情况下，es绑定到localhost hostname，来进行http和内部通信。这对于下载es并简单试用一下，包括日常的开发，都是非常方便的，但是对于生产环境是不行的。如果要组件一个es集群，es实例必须能够通过内部通信协议互相连通，所必须绑定通信到一个外部的接口上。因此如果一个es实例没有绑定通信到外部接口（默认情况下），那么就认为es是处于开发模式下。反之，如果绑定通信到外部接口，那么就是处于生产模式下。

可以通过http.host和transport.host，单独配置http的传输。这就可以配置一个es实例通过http可达，但是却不触发生产模式。

因为有时用户需要将通信绑定到外部解耦来测试client的调用。对于这种场景，es提供了single-node恢复模式（将discovery.type设置为single-node），配置过后，一个节点会选举自己作为master，而且不会跟其他任何节点组成集群。

如果在生产模式下运行一个single node实例，就可以规避掉启动时检查（不要将通信绑定到外部接口，或者将通信绑定到外部接口，但是设置discovery type为single-node）。在这种场景下，可以设置es.enforce.bootstrap.checks为true（通过jvm参数来设置），来强制bootstrap check的执行。

3、heap size check

如果jvm启动的时候设置的初始队大小和最大堆大小不同，可能会导致es运行期间的暂停，因为jvm堆在系统运行期间可能会改变大小。为了避免这种jvm resize导致的es进程暂停，建议启动jvm时，将初始堆大小和最大堆大小设置的相等。除此之外，如果bootstrap.memory\_lock被启用了，jvm会在启动期间锁定jvm的初始大小。

如果要通过heap size check，就必须合理设置heap size。

默认情况下，es的jvm堆的最小和最大大小都是2g。如果在生产环境中使用，应该配置合理的heap size确保es有足够的堆内存可以使用。

在jvm.options中设置的Xms和Xmx会用来分配jvm堆内存带澳。

这些设置的值依赖于服务器上可用的总内存大小。下面是一些最佳实践的建议：

（1）将heap的最小和最大大小设置为一样大

（2）es有更多的heap大小，就有更多的内存用来进行缓存，但是过大的jvm heap可能会导致长时间的gc停顿

（3）不要设置最大heap size超过物理内存的50%，很专业昂才能给核心的file system cache留下足够的内存

（4）不要将Xmx设置超过32GB，否则jvm无法启用compressed oops，将对象指针进行压缩，确认日志里有heap size [1.9gb], compressed ordinary object pointers [true]

（5）更好的选择是，heap size设置的小于zero-based compressed ooops，也就是26GB，但是有时也可以是30GB。通过-XX:+UnlockDiagnosticVMOptions -XX:+PrintCompressedOopsMode开启对应，确认有heap address: 0x000000011be00000, size: 27648 MB, zero based Compressed Oops，而不是heap address: 0x0000000118400000, size: 28672 MB, Compressed Oops with base: 0x00000001183ff000

（6）在jvm.options文件中，可以通过如下方式来配置heap size

-Xms2g

-Xmx2g

（7）也可以通过ES\_JAVA\_OPTS环境变量来设置heap size

ES\_JAVA\_OPTS="-Xms2g -Xmx2g"

4、file descriptor check

file descriptor是unix操作系统的一种数据结构，用来track打开的文件。在unix操作系统中，所有东西都是file。比如，file可以是物理文件，虚拟文件，或者网络socket。es需要大量的file descriptor，比如说每个shard都由多个segment和其他文件组成，还有跟其他节点之间的网络通信连接。

因为es要使用大量的file descriptor，所以如果file descriptor耗尽的话，会是一场灾难，甚至可能会导致数据丢失。尽量给es的file descriptor提升到65536，甚至更高。

可以在/etc/security/limits.conf中，设置nofile为65536

GET \_nodes/stats/process?filter\_path=\*\*.max\_file\_descriptors

可以用上面这行代码检查每个node上的file descriptor数量

lucene会使用大量的文件，同时es也会使用大量的socket在节点间和client间进行通信，这些都是需要大量的file descriptor的。但是通常来说，现在的linux操作系统，都是给每个进程默认的1024个file descriptor的，这对于一个es进程来说是远远不够的。

我们需要将es进程的file descriptor增加到非常非常大，比如说65535个。一般需要根据我们的操作系统的文档来查看如何设置file descriptor。然后可以直接对es集群查看GET，来确认file descriptor的数量：

{

"cluster\_name": "elasticsearch",

"nodes": {

"nLd81iLsRcqmah-cuHAbaQ": {

"timestamp": 1471516160318,

"name": "Marsha Rosenberg",

"transport\_address": "127.0.0.1:9300",

"host": "127.0.0.1",

"ip": [

"127.0.0.1:9300",

"NONE"

],

"process": {

"timestamp": 1471516160318,

"open\_file\_descriptors": 155,

"max\_file\_descriptors": 10240,

"cpu": {

"percent": 0,

"total\_in\_millis": 25084

},

"mem": {

"total\_virtual\_in\_bytes": 5221900288

}

}

}

}

}

5、memory lock check

如果jvm进行一个major gc的话，那么就会涉及到heap中的每一个内存页，此时如果任何一个内存页被swap到了磁盘上，那么此时就会被swap回内存中。这就会导致很多的磁盘读写开销，而这些磁盘读写开销如果节省下来，可以让es服务更多的请求。有很多方法可以配置系统禁止swap。其中一种方法就是让jvm去lock heap内存在物理内存中，设置bootstrap.memory\_lock即可。

GET \_nodes?filter\_path=\*\*.mlockall

检查一下，mlockall是否开启，如果是false，那么说明lock memory失败了，而且日志里可能会有unable to lock jvm memory的字样

可能就是因为运行es的用户没有lock memory的权限，此时就需要进行授权

/etc/security/limits.conf

设置memlock为unlimited即可完成授权

另外一个原因导致lock memory失败，可能是因为临时目录，/tmp用noexec option来mount了

那么就需要设置ES\_JAVA\_OPTS，export ES\_JAVA\_OPTS="$ES\_JAVA\_OPTS -Djava.io.tmpdir=/path/to/temp/dir"

或者在jvm.options中设置这个参数

6、maximum number of thread check

es会将每个请求拆分成多个stage，然后将stage分配到不同的线程池中去执行。在es中有多个线程池来执行不同的任务。所以es会创建许多的线程。最大线程数量的检查会确保说，es实例有权限去创建足够的线程。如果要通过这个检查，必须允许es进程能够创建超过2048个线程。

/etc/security/limits.conf，在这个文件中，用nproc来设置

7、maximum size virtual memory check

es使用mmap来将索引映射到es的address space中，这可以让jvm heap外但是内存中的索引数据，可以有非常告诉的读写速度。因此es需要拥有unlimited address space。最大虚拟内存大小的检查，会要求es进程有unlimited address space。

/etc/security/limits.conf，设置as为unlimited

8、maximum map count check

要高效使用mmap的话，es同样要求创建许多memory-mapped area。因此要求linux内核允许进程拥有至少262144个memory-mapped area，需要通过sysctl设置vm.max\_map\_count至少超过262144。

9、client jvm check

jvm有两种模式，client jvm和server jvm。不同的jvm会用不同的编译器来从java源码生成可执行机器代码。client jvm被优化了来减少startup time和内存占用，server jvm被优化了来最大化性能。两种jvm之间的性能区别是很明显的。client jvm check会确保es没有运行在client jvm下。必须使用server jvm模式来启动es，而server jvm是默认的。

10、use serial collector check

针对不同的工作负载，jvm提供了不同的垃圾回收器。串行化垃圾回收期对于单cpu机器或者小内存，是很有效的。但是对于es来说，用串行化垃圾回收器，会成为一场性能上的灾难。因此这个check会确保es没有被配置使用串行化垃圾回收器。es默认的就是cms垃圾回收器。

11、system call filter check

es会根据不同的操作系统来安装system call filter，用来阻止执行作为defense机制的fork相关system call，进而避免任意代码执行的攻击。这个check会检查是否允许system call filter，然后安装这些system call filter。避免bootstrap.system\_call\_filter设置为false。

12、OnError and OnOutOfMemoryError check

jvm参数，OnError和OnOutOfMemoryError允许在jvm遇到了fatal error或者是OutOfMemoryErro的时候，执行我们预定义的命令。然而，默认情况下，es system call filter是启用的，这些filter是阻止forking操作的。因此，用OnError和OnOutOfMemroyError和system call filter是不兼容的。这个check会检查，如果启用了system call filter，还设置了这两个jvm option，那么就不能启动。所以不要在jvm option中设置这两个参数。

13、early-access check

jdk提供了early-access快照，为即将到来的版本。这些版本不适合用作生产环境。这个check会检查有没有使用jdk的early-access快照版本。我们应该用jdk稳定版本，而不是试用版本。

14、G1 GC check

jdk 8的jvm早期版本中的g1 gc，有已知的问题可能导致索引破损。在JDK 8u40之前的版本都有这个问题。这个check会检查是否使用了那种早期的JDk版本。

## ES 生产集群部署-各个节点一 daemon 模式运行以及优雅关闭

1、以daemon模式运行

在生产环境中，会使用daemon进程的方式来启动es，而不是直接采用前台进程的方式来启动es，具体命令如下

./bin/elasticsearch -d -p pid

上面命令中的-d option用来指定es以daemon进程方式启动，并且-p option指定将进程id记录在指定文件中

es启动后，日志信息可以在ES\_HOME/logs目录中查看

此外，启动es进程的时候，还可以直接覆盖一些配置，使用-E即可，如下面的命令，通常用于调试集群参数时，方便快速调节参数，查看效果

（1）log4j的配置不能有空格

（2）创建专门运行elasticsearch的用户，并授权

像我之前讲课，为了方便，全都是用root用户在做各种操作，但是实际生产环境中，大家应该都知道，root都是那些运维人员的权限

es其实是禁止用root用户去启动es进程的，那么可以加一个配置来允许用root去启动，但是还是算了吧

adduser elasticsearch

passwd elasticsearch

chown -R elasticsearch /usr/local/elasticsearch

chown -R elasticsearch /var/log/elasticsearch

chown -R elasticsearch /var/data/elasticsearch

chown -R elasticsearch /var/plugin/elasticsearch

chown -R elasticsearch /etc/elasticsearch

chown -R elasticsearch /usr/local/tmp

（3）修改/etc/security/limits.conf中的用户为elasticsearch，而不是root

（4）加入memlock的soft unlimited

（5）path.plugins失效，删除这一行配置

（6）jvm.options看来还是用的老的目录中的配置文件

（7）将es的bin加入环境变量PATH中

（8）切换到elasticsearch用户来启动es进程

su elasticsearch

elasticsearch -d -Epath.conf=/etc/elasticsearch

2、访问es

一般建议在管理机上安装一个curl工具，可以手工发送rest api请求

可以对启动了es的节点的9200端口，发送一个GET /请求，可以看看es是否启动成功

curl -XGET elasticsearch02:9200

curl -XGET elasticsearch02:9200/\_cat/nodes?v

3、停止es

优雅的关闭es，可以确保es关闭的很干净，并且优雅关闭资源。举例来说，如果node在一个合理的顺序下关闭了，首先会将自己从cluster中优雅移除，fsync translog日志到磁盘中去，然后执行其他相关的cleanup活动。

如果我们将es用service的方式来运行，那么可以通过server管理功能来停止es。

如果我们是直接启动es的，可以control-C停止es，或者是发送SEGTERM信号给es进程

jps | grep Elasticsearch

kill -SIGTERM 15516

如果es发生了fatal error，类似out of memory error，代码bug，或者io error，等等

当es发现jvm有一个fatal error，就会尝试记录在log里面，然后尝试去停止jvm。此时es是不会按照优雅关闭的模式去执行的，而是会直接关闭，并且返回一个错误码

JVM internal error 128

Out of memory error 127

Stack overflow error 126

Unknown virtual machine error 125

Serious I/O error 124

Unknown fatal error 1

## ES 生产集群备份恢复-hadoop hdfs 分布式文件存储系统

hadoop hdfs是什么的同学，简单的介绍，不会花太多时间，hadoop当前大数据领域的事实上的一个标准

hadoop hdfs，提供的是分布式的文件存储，数据存储

hadoop yarn，提供的是分布式的资源调度

hadoop mapreduce，提供的是分布式的计算引擎，跑在yarn上面的，由yarn去做资源调度

hadoop hive，提供的是分布式的数据仓库引擎，基于mapreduce

hadoop hbase，提供的是分布式的NoSQL数据库，基于hdfs去做的

1、使用课程提供的hadoop-2.7.1.tar.gz，使用WinSCP上传到CentOS的/usr/local目录下。

2、将hadoop包进行解压缩：tar -zxvf hadoop-2.7.1.tar.gz

3、对hadoop目录进行重命名：mv hadoop-2.7.1 hadoop

4、配置hadoop相关环境变量

vi .bashrc

export HADOOP\_HOME=/usr/local/hadoop

export PATH=$HADOOP\_HOME/bin:$HADOOP\_HOME/sbin

source .bashrc

5、在/usr/local目录下创建data目录

6、修改配置文件

core-site.xml

<property>

<name>fs.defaultFS</name>

<value>hdfs://elasticsearch01:9000</value>

</property>

hdfs-site.xml

<property>

<name>dfs.namenode.name.dir</name>

<value>/usr/local/data/namenode</value>

</property>

<property>

<name>dfs.datanode.data.dir</name>

<value>/usr/local/data/datanode</value>

</property>

yarn-site.xml

<property>

<name>yarn.resourcemanager.hostname</name>

<value>elasticsearch01</value>

</property>

mapred-site.xml

<property>

<name>mapreduce.framework.name</name>

<value>yarn</value>

</property>

slaves

elasticsearch01

elasticsearch02

elasticsearch03

在另外两台机器上部署

1、使用scp命令将elasticsearch01上面的hadoop安装包和.bashrc配置文件都拷贝过去。

2、要记得对.bashrc文件进行source，以让它生效。

3、记得在另外两台机器的/usr/local目录下创建data目录。

启动hdfs集群

su elasticsearch

chown -R elasticsearch /usr/local/hadoop

chown -R elasticsearch /usr/local/data

1、格式化namenode：在elasticsearch01上执行以下命令hdfs namenode -format

2、启动hdfs集群：start-dfs.sh

3、验证启动是否成功：jps、50070端口

elasticsearch01：namenode、datanode、secondarynamenode

elasticsearch02：datanode

elasticsearch03：datanode

## ES 生产集群备份恢复-snapshort+hdfs 进行数据备份

1、es集群数据备份

任何一个存储数据的软件，都需要定期的备份我们的数据。es replica提供了运行时的高可用保障机制，可以容忍少数节点的故障和部分数据的丢失，但是整体上却不会丢失任何数据，而且不会影响集群运行。但是replica没法进行灾难性的数据保护，比如说机房彻底停电，所有机器全部当即，等等情况。对于这种灾难性的故障，我们就需要对集群中的数据进行备份了，集群中数据的完整备份。

要备份集群数据，就要使用snapshot api。这个api会将集群当前的状态和数据全部存储到一个外部的共享目录中去，比如NAS，或者hdfs。而且备份过程是非常智能的，第一次会备份全量的数据，但是接下来的snapshot就是备份两次snapshot之间的增量数据了。数据是增量进入es集群或者从es中删除的，那么每次做snapshot备份的时候，也会自动在snapshot备份中增量增加数据或者删除部分数据。因此这就意味着每次增量备份的速度都是非常快的。

如果要使用这个功能，我们需要有一个预先准备好的独立于es之外的共享目录，用来保存我们的snapshot备份数据。es支持多种不同的目录类型：shared filesystem，比如NAS；Amazon S3；hdfs；Azure Cloud。不过对于国内的情况而言，其实NAS应该很少用，一般来说，就用hdfs会比较多一些，跟hadoop这种离线大数据技术栈整合起来使用。

2、创建备份仓库

（1）创建和查询仓库的命令

PUT \_snapshot/my\_backup

{

"type": "fs",

"settings": {

"location": "/mount/backups/my\_backup"

}

}

这里用了shared filesystem作为仓库类型，包括了仓库名称以及仓库类型是fs，还有仓库的地址。这个里面就包含了仓库的一些必要的元数据了。可能还有其他的一些参数可以配置，主要是基于我们的node和网络的性能来配置。max\_snapshot\_bytes\_per\_sec，这个参数用于指定数据从es灌入仓库的时候，进行限流，默认是20mb/s。max\_restore\_bytes\_per\_sec，这个参数用于指定数据从仓库中恢复到es的时候，进行限流，默认也是20mb/s。假如说网络是非常快速的，那么可以提高这两个参数的值，可以加快每次备份和恢复的速度，比如下面：

POST \_snapshot/my\_backup/

{

"type": "fs",

"settings": {

"location": "/mount/backups/my\_backup",

"max\_snapshot\_bytes\_per\_sec" : "50mb",

"max\_restore\_bytes\_per\_sec" : "50mb"

}

}

创建一个仓库之后，就可以查看这个仓库的信息了：GET /\_snapshot/my\_backup，或者是查看所有的仓库，GET /\_snapshot/\_all。可能返回如下的信息：

{

"my\_backup": {

"type": "fs",

"settings": {

"compress": true,

"location": "/mount/backups/my\_backup"

}

}

}

（2）基于hdfs创建仓库

但是其实如果在国内使用es的话，还是建议跟hadoop生态整合使用，不要用那种shared filesystem。可以用hadoop生态的hdfs分布式文件存储系统。首先先要安装repository-hdfs的插件：bin/elasticsearch-plugin install repository-hdfs，必须在每个节点上都安装，然后重启整个集群。

kill -SIGTERM 15516

su elasticsearch

elasticsearch -d -Epath.conf=/etc/elasticsearch

curl -XGET elasticsearch02:9200/\_cat/nodes?v

在3个hdfs node上，都加入hdfs-site.xml，禁止权限检查，如果要修改这个配置文件，要先在/usr/local/hadoop/sbin，运行./stop-dfs.sh，停止整个hdfs集群，然后在3个node上，都修改hdfs-site.xml，加入下面的配置，禁止权限的检查

<property>

<name>dfs.permissions</name>

<value>false</value>

</property>

hdfs snapshot/restore plugin是跟最新的hadoop 2.x整合起来使用的，目前是hadoop 2.7.1。所以如果我们使用的hadoop版本跟这个es hdfs plugin的版本不兼容，那么考虑在hdfs plugin的文件夹里，将hadoop相关jar包都替换成我们自己的hadoop版本对应的jar包。即使hadoop已经在es所在机器上也安装了，但是为了安全考虑，还是应该将hadoop jar包放在hdfs plugin的目录中。

安装好了hdfs plugin之后，就可以创建hdfs仓库了，用如下的命令即可：

curl -XGET 'http://localhost:9200/\_count?pretty' -d '

{

"query": {

"match\_all": {}

}

}

'

curl -XPUT 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository2' -d '

{

"type": "hdfs",

"settings": {

"uri": "hdfs://elasticsearch02:9000/",

"path": "elasticsearch/respositories/my\_hdfs\_repository",

"conf.dfs.client.read.shortcircuit": "false",

"max\_snapshot\_bytes\_per\_sec" : "50mb",

"max\_restore\_bytes\_per\_sec" : "50mb"

}

}'

（3）验证仓库

在课程演示中，最好都是用root用户去演示，一般来说就够了，因为在不同的公司里，你可能linux用户管理，权限，都不太一样

专门去建一套用户和授权去演示，不太合适

如果一个仓库被创建好之后，我们可以立即去验证一下这个仓库是否可以在所有节点上正常使用。verify参数都可以用来做这个事情，比如下面的命令。这个命令会返回一个node列表，证明那些node都验证过了这个仓库是ok的，可以使用的

curl -XPOST 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/\_verify'

先停止整个es集群，然后在3个节点上，都加入下面的配置，然后用elasticsearch账号重启整个es集群

/usr/local/elasticsearch/plugins/repository-hdfs/plugin-security.policy

permission java.lang.RuntimePermission "accessDeclaredMembers";

permission java.lang.RuntimePermission "getClassLoader";

permission java.lang.RuntimePermission "shutdownHooks";

permission java.lang.reflect.ReflectPermission "suppressAccessChecks";

permission javax.security.auth.AuthPermission "doAs";

permission javax.security.auth.AuthPermission "getSubject";

permission javax.security.auth.AuthPermission "modifyPrivateCredentials";

permission java.security.AllPermission;

permission java.util.PropertyPermission "\*", "read,write";

permission javax.security.auth.PrivateCredentialPermission "org.apache.hadoop.security.Credentials \* \"\*\"", "read";

/usr/local/elasticsearch/config/jvm.options

-Djava.security.policy=file:////usr/local/elasticsearch/plugins/repository-hdfs/plugin-security.policy

3、对索引进行snapshotting备份

（1）对所有open的索引进行snapshotting备份

一个仓库可以包含多分snapshot，每个snapshot是一部分索引的备份数据，创建一份snapshot备份时，我们要指定要备份的索引。比如下面这行命令：PUT \_snapshot/my\_hdfs\_repository/snapshot\_1，这行命令就会将所有open的索引都放入一个叫做snapshot\_1的备份，并且放入my\_backup仓库中。这个命令会立即返回，然后备份操作会被后台继续进行。如果我们不希望备份操作以后台方式运行，而是希望在前台发送请求时等待备份操作执行完成，那么可以加一个参数即可，比如下面这样：PUT \_snapshot/my\_backup/snapshot\_1?wait\_for\_completion=true。

curl -XPUT 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1'

（2）对指定的索引进行snapshotting备份

默认的备份是会备份所有的索引，但是有的时候，可能我们不希望备份所有的索引，有些可能是不重要的数据，而且量很大，没有必要占用我们的hdfs磁盘资源，那么可以指定备份少数重要的数据即可。此时可以使用下面的命令去备份指定的索引：

PUT \_snapshot/my\_backup/snapshot\_2

{

"indices": "index\_1,index\_2",

"ignore\_unavailable": true,

"include\_global\_state": false,

"partial": true

}

ignore\_unavailable如果设置为true的话，那么那些不存在的index就会被忽略掉，不会进行备份过程中。默认情况下，这个参数是不设置的，那么此时如果某个index丢失了，会导致备份过程失败。设置include\_global\_state为false，可以阻止cluster的全局state也作为snapshot的一部分被备份。默认情况下，如果某个索引的部分primary shard不可用，那么会导致备份过程失败，那么此时可以将partial设置为true。

而且snapshotting的过程是增量进行的，每次执行snapshotting的时候，es会分析已经存在于仓库中的snapshot对应的index file，然后仅仅备份那些自从上次snapshot之后新创建的或者有过修改的index files。这就允许多个snapshot在仓库中可以用一种紧凑的模式来存储。而且snapshotting过程是不会阻塞所有的es读写操作的，然而，在snapshotting开始之后，写入index中的数据，是不会反应到这次snapshot中的。每次snapshot除了创建一份index的副本之外，还可以保存全局的cluster元数据，里面包含了全局的cluster设置和template。

每次只能执行一次snapshot操作，如果某个shard正在被snapshot备份，那么这个shard此时就不能被移动到其他node上去，这会影响shard rebalance的操作。只有在snapshot结束之后，这个shard才能够被移动到其他的node上去。

4、查看snapshot备份列表

一旦我们在仓库中备份了一些snapshot之后，就可以查看这些snapshot相关的详细信息了，使用这行命令就可以查看指定的snapshot的详细信息：GET \_snapshot/my\_backup/snapshot\_2，结果大致如下所示。当然也可以查看所有的snapshot列表，GET \_snapshot/my\_backup/\_all。

curl -XGET 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1?pretty'

{

"snapshots" : [

{

"snapshot" : "snapshot\_1",

"uuid" : "x8DXcrp2S0md-BC9ftYZqw",

"version\_id" : 5050099,

"version" : "5.5.0",

"indices" : [

"my\_index"

],

"state" : "SUCCESS",

"start\_time" : "2017-07-08T19:54:54.914Z",

"start\_time\_in\_millis" : 1499543694914,

"end\_time" : "2017-07-08T19:54:56.886Z",

"end\_time\_in\_millis" : 1499543696886,

"duration\_in\_millis" : 1972,

"failures" : [ ],

"shards" : {

"total" : 5,

"failed" : 0,

"successful" : 5

}

}

]

}

5、删除snapshot备份

如果要删除过于陈旧的snapshot备份快照，那么使用下面这行命令即可：DELETE \_snapshot/my\_backup/snapshot\_2。记住，一定要用api去删除snapshot，不要自己手动跑到hdfs里删除这个数据。因为snapshot是增量的，有可能很多snapshot依赖于底层的某一个公共的旧的snapshot segment。但是delete api是理解数据如何增量存储和互相依赖的，所以可以正确的删除那些不用的数据。如果我们自己手工进行hdfs文件删除，可能导致我们的backup数据破损掉，就无法使用了。

curl -XDELETE 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1'

6、监控snapshotting的进度

使用wait\_for\_completion可以在前台等待备份完成，但是实际上也没什么必要，因为可能要备份的数据量特别大，难道还等待1个小时？？看着是不太现实的，所以一般还是在后台运行备份过程，然后使用另外一个监控api来查看备份的进度，首先可以获取一个snapshot ID：GET \_snapshot/my\_backup/snapshot\_3。如果这个snapshot还在备份过程中，此时我们就可以看到一些信息，比如什么时候开始备份的，已经运行了多长时间，等等。然而，这个api用了跟snapshot一样的线程池去执行，如果我们在备份非常大的shard，进度的更新可能会非常之慢。一个更好的选择是用\_status API，GET \_snapshot/my\_backup/snapshot\_3/\_status，这个api立即返回最详细的数据。这里我们可以看到总共有几个shard在备份，已经完成了几个，还剩下几个，包括每个索引的shard的备份进度：

curl -XGET 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1'

7、取消snapshotting备份过程

如果我们想要取消一个正在执行的snapshotting备份过程，比如我们发现备份时间过于长，希望先取消然后在晚上再运行，或者是因为不小心误操作发起了一次备份操作，这个时候就可以运行下面这条命令：DELETE \_snapshot/my\_backup/snapshot\_3。也就是立即删除这个snapshot，这个命令会去取消snapshot的过程，同时将备份了一半的仓库中的数据给删除掉。

curl -XDELETE 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1'

## ES 生产集群备份恢复-基于 snapshort+hdfs+restore进行数据恢复

1、基于snapshot的数据恢复

正经备份，一般来说，是在一个shell脚本里，你用crontab做一个定时，比如每天凌晨1点，就将所有的数据做一次增量备份，当然，如果你的数据量较大，每小时做一次也ok。shell脚本里，就用curl命令，自动发送一个snapshot全量数据的请求。那么这样的话，就会自动不断的去做增量备份。

20170721，做了一次snapshot，snapshot\_20170721

20170722，又做了一次snapshot，snapshot\_20170722

这两次snapshot是有关联关系的，因为第二次snapshot是基于第一次snapshot的数据，去做的增量备份

如果你要做数据恢复，比如说，你自己误删除，不小心将整个index给删除掉了，数据丢了，很简单，直接用最新的那个snapshot就可以了，比如snapshot\_20170722

如果，你是做了一些错误的数据操作，举个例子，今天你的程序有个bug，写入es中的数据都是错误的，需要清洗数据，重新导入

这个时候，虽然最新的snapshot\_20170722，但是也可以手动选择snapshot\_20170721 snapshot，去做恢复，相当于是将数据恢复到20170721号的数据情况，忽略掉20170722号的数据的变更

然后重新去导入数据

如果我们用一些脚本定期备份数据之后，那么在es集群故障，导致数据丢失的时候，就可以用\_restore api进行数据恢复了。比如下面这行命令：POST \_snapshot/my\_hdfs\_repository/snapshot\_1/\_restore。这个时候，会将那个snapshot中的所有数据恢复到es中来，如果snapshot\_1中包含了5个索引，那么这5个索引都会恢复到集群中来。不过我们也可以选择要从snapshot中恢复哪几个索引。

我们还可以通过一些option来重命名索引，恢复索引的时候将其重命名为其他的名称。在某些场景下，比如我们想恢复一些数据但是不要覆盖现有数据，然后看一下具体情况，用下面的命令即可恢复数据，并且进行重命名操作：

POST /\_snapshot/my\_hdfs\_repository/snapshot\_1/\_restore

{

"indices": "index\_1",

"ignore\_unavailable": true,

"include\_global\_state": true,

"rename\_pattern": "index\_(.+)",

"rename\_replacement": "restored\_index\_$1"

}

index\_01

restores\_index\_01

这个restore过程也是在后台运行的，如果要在前台等待它运行完，那么可以加上wait\_for\_completion flag：POST \_snapshot/my\_backup/snapshot\_1/\_restore?wait\_for\_completion=true。

?wait\_for\_completion=true，包括之前讲解的备份，也是一样的，对运维中的自动化shell脚本，很重要，你的shell脚本里，要比如等待它备份完成了以后，才会去执行下一条命令

restore过程只能针对已经close掉的index来执行，而且这个index的shard还必须跟snapshot中的index的shard数量是一致的。restore操作会自动在恢复好一个index之后open这个index，或者如果这些index不存在，那么就会自动创建这些index。如果通过include\_global\_state存储了集群的state，还会同时恢复一些template。

默认情况下，如果某个索引在恢复的时候，没有在snapshot中拥有所有的shard的备份，那么恢复操作就会失败，比如某个shard恢复失败了。但是如果将partial设置为true，那么在上述情况下，就还是可以进行恢复操作得。不过在恢复之后，会自动重新创建足够数量的replica shard。

此外，还可以在恢复的过程中，修改index的一些设置，比如下面的命令：

POST /\_snapshot/my\_backup/snapshot\_1/\_restore

{

"indices": "index\_1",

"index\_settings": {

"index.number\_of\_replicas": 0

},

"ignore\_index\_settings": [

"index.refresh\_interval"

]

}

curl -XDELETE 'http://elasticsearch02:9200/my\_index?pretty'

curl -XGET 'http://elasticsearch02:9200/my\_index/my\_type/1'

curl -XPOST 'http://elasticsearch02:9200/\_snapshot/my\_hdfs\_repository/snapshot\_1/\_restore?pretty'

curl -XGET 'http://elasticsearch02:9200/my\_index/my\_type/1'

解释一下，为什么大家会发现，我们这次集群运维的课程，会大量的用curl命令来操作，作为es集群管理员的话，有的时候可能还是要用curl命令，特别是可能要封装一些自动化的shell脚本，去做一些es的运维操作，比如自动定时备份

可能你要做一次恢复，连带执行很多es的运维命令，可以提前封装一个shell脚本，大量的就是要用curl命令

2、监控restore的进度

从一个仓库中恢复数据，其实内部机制跟从其他的node上恢复一个shard是一样的。如果要监控这个恢复的过程，可以用recovery api，比如：GET restored\_index\_3/\_recovery。如果要看所有索引的恢复进度：GET /\_recovery/。可以看到恢复进度的大致的百分比。结果大致如下所示：

curl -XGET 'http://elasticsearch02:9200/my\_index/\_recovery?pretty'

3、取消恢复过程

如果要取消一个恢复过程，那么需要删除已经被恢复到es中的数据。因为一个恢复过程就只是一个shard恢复，发送一个delete操作删除那个索引即可，比如：DELETE /restored\_index\_3。如果那个索引正在被恢复，那么这个delete命令就会停止恢复过程，然后删除已经恢复的 所有数据。

curl -XDELETE 'http://elasticsearch02:9200/my\_index'

## ES 生产集群版本升级-基于节点依次重启策略进行 5.x 的各个小版本之间的升级

生产集群的时候，版本升级，是不可避免的，es主要的版本，es 2.x，es 5.x，es 1.x

如果你去运维一个es的集群，你要做各个版本之间的升级，你该怎么做。。。

三种策略：

1、面向的场景，就是同一个大版本之间的各个小版本的升级，比如说，我们这一节课，es 5.3升级到es 5.5

2、相邻的两个大版本之间的升级，比如es 2.x升级到es 5.x，es 2.4.3，升级到es 5.5

3、跨了几个大版本之间的升级，就比如说es 1.x升级到es 5.x

每一种场景使用的技术方案都是不一样的

1、es版本升级的通用步骤

（1）看一下最新的版本的breaking changes文档，官网，看一下，每个小版本之间的升级，都有哪些变化，新功能，bugfix

（2）用elasticsearch migration plugin在升级之前检查以下潜在的问题（在老版本的时候可能还会去用，现在新版本这个plugin很少用了）

（3）在开发环境的机器中，先实验一下版本的升级，定一下升级的技术方案和操作步骤的文档，然后先在你的测试环境里，先升级一次，搞一下

（4）对数据做一次全量备份，备份和恢复，最次最次的情况，哪怕是升级失败了，哪怕是你重新搭建一套全新的es

（5）检查升级之后各个plugin是否跟es主版本兼容，升级完es之后，还要重新安装一下你的plugin

es不同版本之间的升级，用的升级策略是不一样的

（1）es 1.x升级到es 5.x，是需要用索引重建策略的

（2）es 2.x升级到es 5.x，是需要用集群重启策略的

（3）es 5.x升级到es 5.y，是需要用节点依次重启策略的

es的每个大版本都可以读取上一个大版本创建的索引文件，但是如果是上上个大版本创建的索引，是不可以读取的。比如说es 5.x可以读取es 2.x创建的索引，但是没法读取es 1.x创建的索引。

2、rolling upgrade（节点依次重启策略）

rolling upgrade会让es集群每次升级一个node，对于终端用户来说，是没有停机时间的。在一个es集群中运行多个版本，长时间的话是不行的，因为shard是没法从一较新版本的node上replicate到较旧版本的node上的。

先部署一个es 5.3.2版本，将配置文件放在外部目录，同时将data和log目录都放在外部，然后插入一些数据，然后再开始下面的升级过程

adduser elasticsearch

passwd elasticsearch

chown -R elasticsearch /usr/local/elasticsearch

chown -R elasticsearch /var/log/elasticsearch

chown -R elasticsearch /var/data/elasticsearch

chown -R elasticsearch /etc/elasticsearch

su elasticsearch

elasticsearch -d -Epath.conf=/etc/elasticsearch

curl -XPUT 'http://localhost:9200/forum/article/1?pretty' -d '

{

"title": "first article",

"content": "this is my first article"

}'

（1）禁止shard allocation

停止一个node之后，这个node上的shard全都不可用了，此时shard allocation机制会等待一分钟，然后开始shard recovery过程，也就是将丢失掉的primary shard的replica shard提升为primary shard，同时创建更多的replica shard满足副本数量，但是这个过程会导致大量的IO操作，是没有必要的。因此在开始升级一个node，以及关闭这个node之前，先禁止shard allocation机制：

curl -XPUT 'http://localhost:9200/\_cluster/settings?pretty' -d '

{

"persistent": {

"cluster.routing.allocation.enable": "none"

}

}'

（2）停止非核心业务的写入操作，以及执行一次flush操作

可以在升级期间继续写入数据，但是如果在升级期间一直写入数据的话，可能会导致重启节点的时候，shard recovery的时间变长，因为很多数据都是translog里面，没有flush到磁盘上去。如果我们暂时停止数据的写入，而且还进行一次flush操作，把数据都刷入磁盘中，这样在node重启的时候，几乎没有什么数据要从translog中恢复的，重启速度会很快，因为shard recovery过程会很快。用下面这行命令执行flush：POST \_flush/synced。但是flush操作是尽量执行的，有可能会执行失败，如果有大量的index写入操作的话。所以可能需要多次执行flush，直到它执行成功。

curl -XPOST 'http://localhost:9200/\_flush/synced?pretty'

（3）停止一个node然后升级这个node

在完成node的shard allocation禁用以及flush操作之后，就可以停止这个node。

如果你安装了一些插件，或者是自己设置过jvm.options文件的话，需要先将/usr/local/elasticsearch/plugins拷贝出来，作为一个备份，jvm.options也拷贝出来

将老的es安装目录删除，然后将最新版本的es解压缩，而且要确保我们绝对不会覆盖config、data、log等目录，否则就会导致我们丢失数据、日志、配置文件还有安装好的插件。

kill -SIGTERM 15516

（4）升级plugin

可以将备份的plugins目录拷贝回最新解压开来的es安装目录中，包括你的jvm.options

自己去官网，找到各个plugin的git地址，git地址上，都有每个plugin version跟es version之间的对应关系。要检查一下所有的plugin是否跟要升级的es版本是兼容的，如果不兼容，那么需要先用elasticsearch-plugin脚本重新安装最新版本的plugin。

（5）启动es node

接着要注意在启动es的时候，在命令行里用-Epath.conf= option来指向一个外部已经配置好的config目录。这样的话最新版的es就会复用之前的所有配置了，而且也会根据配置文件中的地址，找到对应的log、data等目录。然后再日志中查看这个node是否正确加入了cluster，也可以通过下面的命令来检查：GET \_cat/nodes。

elasticsearch -d -Epath.conf=/etc/elasticsearch

（6）在node上重新启用shard allocation

一旦node加入了cluster之后，就可以重新启用shard allocation

curl -XPUT 'http://localhost:9200/\_cluster/settings?pretty' -d '

{

"persistent": {

"cluster.routing.allocation.enable": "all"

}

}'

（7）等待node完成shard recover过程

我们要等待cluster完成shard allocation过程，可以通过下面的命令查看进度：GET \_cat/health。一定要等待cluster的status从yellow变成green才可以。green就意味着所有的primary shard和replica shard都可以用了。

curl -XGET 'http://localhost:9200/\_cat/health?pretty'

在rolling upgrade期间，primary shard如果分配给了一个更新版本的node，是一定不会将其replica复制给较旧的版本的node的，因为较新的版本的数据格式跟较旧的版本是不兼容的。但是如果不允许将replica shard复制给其他node的话，比如说此时集群中只有一个最新版本的node，那么有些replica shard就会是unassgied状态，此时cluster status就会保持为yellow。此时，就可以继续升级其他的node，一旦其他node变成了最新版本，那么就会进行replica shard的复制，然后cluster status会变成green。

如果没有进行过flush操作的shard是需要一些时间去恢复的，因为要从translog中恢复一些数据出来。可以通过下面的命令来查看恢复的进度：GET \_cat/recovery。

（8）重复上面的步骤，直到将所有的node都升级完成

## ES 生产集群版本升级-基于集群整体重启策略进行2.x到 5.x 的大版本升级

滚动升级策略，集群，集群里面有多个节点，一个节点一个节点的重启和升级

如果是大版本之间的升级，集群重启策略，要先将整个集群全部停掉，如果采取滚动升级策略的话，可能导致说，一个集群内，有些节点是es 5.5，有些节点是es 2.4.3，这样的话是可能会有问题的

升级的过程，其实是跟之前的一模一样的

es在进行重大版本升级的时候，一般都需要采取full cluster restart的策略，重启整个集群来进行升级。rolling upgrade在重大版本升级的时候是不合适的。

执行一个full cluster restart升级的过程如下：

我们先停掉之前的es 5.5，删除所有相关的目录，然后安装一个es 2.4.3，再将其升级到es 5.5

chown -R elasticsearch /usr/local/elasticsearch

chown -R elasticsearch /var/log/elasticsearch

chown -R elasticsearch /var/data/elasticsearch

chown -R elasticsearch /etc/elasticsearch

su elasticsearch

elasticsearch -d -Dpath.conf=/etc/elasticsearch

kill -SIGTERM 15516

curl -XPUT 'http://localhost:9200/forum/article/1?pretty' -d '

{

"title": "first article",

"content": "this is my first article"

}'

（1）禁止shard allocation

我们停止一个node时，可能导致部分replica shard死掉了，此时shard allocation机制会立即在其他节点上分配一些replica shard过去。如果是停止node导致primary shard死掉了，会将其他node上的replica shard提升为primary shard，同理会给其复制足够的replica shard，保持replica副本数量。但是这回导致大量的IO开销。我们首先得先禁止这个机制：

curl -XPUT 'http://localhost:9200/\_cluster/settings?pretty' -d '

{

"persistent": {

"cluster.routing.allocation.enable": "none"

}

}'

（2）执行一次flush操作

我们最好是停止接受新的index写入操作，并且执行一次flush操作，确保数据都fsync到磁盘上。这样的话，确保没有数据停留在内存和WAL日志中。shard recovery的时间就会很短。

curl -XPOST 'http://localhost:9200/\_flush/synced?pretty'

此时，最好是执行synced flush操作，因为我们最好是确保说flush操作成功了，再执行下面的操作

如果flush操作报错了，那么可以反复多执行几次

（3）关闭和升级所有的node

如果是将es 2.x版本升级到es 5.x版本，唯一的区别就在这里开始了，先将整个集群中所有的节点全部停止

将最新版本的es解压缩替代之前的es安装目录之前，一定要记得先将plugins做个备份

将集群上所有node上的es服务都给停止，然后按照rolling upgrade中的步骤对集群中所有的node进行升级

将所有的节点全部停掉，将所有的node全部替换为最新版本的es安装目录

（4）升级plugin

最新版的es解压开来以后，就可以看看，可以去做一个plugin的升级

es plugin的版本是跟es版本相关联的，因此必须使用elasticsearch-plugin脚本来安装最新的plugin版本

（5）启动cluster集群

如果我们有专门的master节点的话，就是那些将node.master设置为true的节点（默认都是true，都有能力作为master节点），而且node.data设置为false，那么就先将master node启动。等待master node组建成一个cluster之后，这些master node中会选举一个正式的master node出来。可以在log中检查master的选举。

只要minimum number of master-eligible nodes数量的node发现了彼此，他们就会组成一个cluster，并且选举出来一个master。从这时开始，可以监控到加入cluster的node。

依次将所有的node重新启动起来

elasticsearch -d -Epath.conf=/etc/elasticsearch

如果是将es 2.x升级到es 5.x，记得将log4j.properties拷贝到你外部的目录中去，而且还要重新做目录的权限的更改

curl -XGET 'http://localhost:9200/\_cat/health?pretty'

curl -XGET 'http://localhost:9200/\_cat/nodes?pretty'

（6）等待cluster状态变成yellow

只要每个ndoe都加入了cluster，就会开始对primary shard进行receover过程，就是看有没有数据在WAL日志中的，给恢复到内存里。刚开始的话，\_cat/health请求会反馈集群状态是red，这意味着不是所有的primary shard都被分配了。

只要每个node发现了自己本地的shard之后，集群status就会变成yellow，意味着所有的primary shard都被发现了，但是并不是所有的replica shard都被分配了。

（7）重新启用allocation

直到所有的node都加入了集群，再重新启用shard allocation，可以让master将replica分配给那些本地已经有replica shard的node上。

curl -XPUT 'http://localhost:9200/\_cluster/settings?pretty' -d '

{

"persistent": {

"cluster.routing.allocation.enable": "all"

}

}'

cluster这个时候就会开始将replica shard分配给data node。此时可以恢复index和search操作，不过最好还是等待replica shard全部分配完之后，再去恢复读写操作。

我们可以通过下面的api来监控这个过程

GET \_cat/health

GET \_cat/recovery

如果\_cat/health中的status列变成了green，那么所有的primary和replica shard都被成功分配了

## ES 生产进群版本升级-基于索引重建策略进行 1.x 到 5.x 跨两个版本升级

es只能使用上一个大版本创建的索引。举例来说，es 5.x可以使用es 2.x中的索引，但是不能使用es 1.x中的索引。

es 5.x如果使用过于陈旧的索引去启动，就会启动失败

如果我们在运行es 2.x集群，但是索引是从2.x之前的版本创建的，那么在升级到es 5.x之前，我们需要删除旧索引，或者reindex这些索引，用reindex in place的策略。

如果我们在运行es 1.x的集群，有两个选择，首先升级到es 2.4.x，然后reindex所有的旧索引，用reindex in place的策略，接着升级到es 5.x。创建一个新的5.x集群，然后使用reindex-from-remote直接从es 1.x集群中将索引倒入5.x集群中。

同时运行一个es 1.x的集群，同时也运行一个es 5.x的集群，然后用reindex功能，将es 1.x中的所有数据都导入到es 5.x集群中

elasticsearch -d -Dpath.conf=/etc/elasticsearch

kill -SIGTERM 15516

chown -R elasticsearch /usr/local/elasticsearch-1.7.4

chown -R elasticsearch /usr/local/elasticsearch-5.5.0

curl -XPUT 'http://localhost:9200/forum/article/1?pretty' -d '

{

"title": "first article",

"content": "this is my first article"

}'

（1）reindex in place

将1.x中的索引reindex最简单的方法，就是用elasticsearch migration plugin去做reindex。但是首先需要先升级到es 2.3.x或2.4.x。

migration plugin中提供的reindex工具会执行以下操作：

创建新的索引，但是会将es版本号拼接到索引名称上，比如my\_index-2.4.1，从旧的索引中拷贝mapping和setting。禁止新索引的refresh，并且将replica数量设置为0.主要是为了更高效的reindex。

将旧索引设置为只读，不允许新的数据写入旧索引中

从旧索引中，将所有的数据reindex到新索引中

对新索引的refresh\_interval和number\_of\_replicas的值重新设置为旧索引中的值，并且等待索引变成green

将旧索引中存在的alias添加到新索引中

删除旧的索引

给新索引添加一个alia，使用旧索引的名称，比如将my\_index设置为my\_index-2.4.1的alia

此时，就可以有一份新的2.x的索引，可以在5.x中使用

（2）upgrading with reindex-from-remote

如果在运行1.x cluster，并且想要直接迁移到5.x，而不是先迁移到2.x，那么需要进行reindex-from-remote操作

es包含了向后兼容性的代码，从而允许上一个大版本的索引可以直接在这个版本中使用。如果要直接从1.x升级到5.x，我们就需要自己解决向后兼容性的问题。

首先我们需要先建立一个新的5.x的集群。5.x集群需要能够访问1.x集群的rest api接口。

对于每个我们想要迁移到5.x集群的1.x的索引，需要做下面这些事情：

在5.x中创建新的索引，以及使用合适的mapping和setting，将refresh\_interval设置为-1，并且设置number\_of\_replica为0，主要是为了更快的reindex。

用reindex from remote的方式，在两个集群之间迁移index数据

curl -XPOST 'http://localhost:9201/\_reindex?pretty' -d '

{

"source": {

"remote": {

"host": "http://localhost:9200"

},

"index": "forum"

},

"dest": {

"index": "forum"

}

}'

remote cluster必须显示在elasticsearch.yml中列入白名单中，使用reindex.remote.whitelist属性

reinde过程中会使用的默认的on-heap buffer最大大小是100mb，如果要迁移的数据量很大，需要将batch size设置的很小，这样每次同步的数据就很少，使用size参数。还可以设置socket\_timeout和connect\_timeout，比如下面：

POST \_reindex

{

"source": {

"remote": {

"host": "http://otherhost:9200",

"socket\_timeout": "1m",

"connect\_timeout": "10s"

},

"index": "source",

"size": 10,

"query": {

"match": {

"test": "data"

}

}

},

"dest": {

"index": "dest"

}

}

如果在后台运行reindex job，就是将wait\_for\_completion设置为false，那么reindex请求会返回一个task\_id，后面可以用来监控这个reindex progress的进度，GET \_tasks/TASK\_ID

一旦reindex完成之后，可以将refresh\_interval和number\_of\_replicas设置为正常的数值，比如30s和1

一旦新的索引完成了replica操作，就可以删除旧的index了

## Elasticsearch 生产集群中的索引管理（一）

1、创建索引

（1）创建索引的语法

用settings给这个索引在创建时可以添加一些设置，还有可以初始化一些type的mapping

curl -XPUT 'http://elasticsearch02:9200/twitter?pretty' -d '

{

"settings" : {

"index" : {

"number\_of\_shards" : 3,

"number\_of\_replicas" : 2

}

},

"mappings" : {

"type1" : {

"properties" : {

"field1" : { "type" : "text" }

}

}

}

}'

（2）索引创建返回消息的解释

默认情况下，索引创建命令会在每个primary shard的副本开始进行复制以后，或者是请求超时以后，返回一个响应消息，类似下面这样的。其中acknowledged表明了这个索引有没有创建成功，shards\_acknowledged表明了每个primary shard有没有足够数量的replica开始进行复制了。有可能这两个参数会为false，但是索引依然可以创建成功。因为这些参数仅仅是表明在请求超时之前，那两个行为有没有成功，也有可能请求超时了，在超时前都没成功，但是超时后在es server端还是都执行了。如果acknoledged是false，那么就可能是超时了，此时接受到响应消息的时候，cluster state都还没变更，没有加入新创建的index，但是也许之后还是会创建这个index。如果shards\_acknowledged是false，那么可能在primary shard进行副本copy之前，就timeout了，但是此时也许index创建成功了，而且cluster state已经加入了新创建的index。

{

"acknowledged": true,

"shards\_acknowledged": true

}

2、删除索引

curl -XDELETE 'http://elasticsearch02:9200/twitter?pretty'

删除索引中的一个type

3、查询索引设置信息

curl -XGET 'http://elasticsearch02:9200/twitter?pretty'

4、打开/关闭索引

curl -XPOST 'http://elasticsearch02:9200/twitter/\_close?pretty'

curl -XPOST 'http://elasticsearch02:9200/twitter/\_open?pretty'

curl -XPUT 'http://elasticsearch02:9200/twitter/type1/1?pretty' -d '

{

"field1": "1"

}'

如果关闭了一个索引之后，那么这个索引是不会带来任何的性能开销了，只要保留这个索引的元数据即可，然后对这个索引的读写操作都不会成功。一个关闭的索引可以接着再打开，打开以后会进行shard recovery过程。

比如说你在做一些运维操作的时候，现在你要对某一个索引做一些配置，运维操作，修改一些设置，关闭索引，不允许写入，成功以后再打开索引

5、压缩索引

shrink命令可以将一个已有的索引压缩成一个新的索引，同时primary shard会更少。因为以前提到过，primary shard因为涉及到document的hash路由问题，所以是不允许修改的。但是如果要减少index的primary shard，可以用shrink命令来压缩index。但是压缩后的shard数量必须可以被原来的shard数量整除。举例来说，一个有8个primary shard的index可以被压缩成4个，2个，或者1个primary shard的index。

压缩索引，是这样啊，如果你的索引中本来比如是要保留7天的数据，那么给了10个shard，但是现在需求变了，这个索引只要保留3天的数据就可以了，那么数据量变小了，就不需要10个shard了，就可以做shrink操作，5个shard。

shrink命令的工作流程如下：

（1）首先，它会创建一个跟source index的定义一样的target index，但是唯一的变化就是primary shard变成了指定的数量

（2）接着它会将source index的segment file直接用hard-link的方式连接到target index的segment file，如果操作系统不支持hard-link，那么就会将source index的segment file都拷贝到target index的data dir中，会很耗时。如果用hard-link会很快

（3）最后，会将target index进行shard recovery恢复

如果要shrink index，那么这个index必须先被标记为read only，而且这个index的每个shard的某一个copy，可以是primary或者是replica，都必须被复制到一个节点上去。默认情况下，index的每个shard有可能在不同机器上的，比如说，index有5个shard，shard0和shard1在机器1上，shard2、shard3在机器2上，shard4在机器3上。现在还得把shard0，shard1，shard2，shard3，shard4全部拷贝到一个同一个机器上去，但是可以是shard0的replica shard。而且每个primary shard都必须存在。可以通过下面的命令来完成。其中index.routing.allocation.require.\_name必须是某个node的名称，这个都是可以自己设置的。

curl -XPUT 'http://elasticsearch02:9200/twitter/\_settings?pretty' -d '

{

"settings": {

"index.routing.allocation.require.\_name": "node-elasticsearch-02",

"index.blocks.write": true

}

}'

这个命令会花费一点时间将source index每个shard的一个copy都复制到指定的node上去，可以通过GET \_cat/recovery?v命令来追踪这个过程的进度。

等上面的shard copy relocate过程结束之后，就可以shrink一个index，用下面的命令即可：POST my\_source\_index/\_shrink/my\_target\_index。如果target index被添加进了cluster state之后，这个命令就会立即返回，不是等待shrink过程完成之后才返回的。当然还可以用下面的命令来shrink的时候修改target index的设置，在settings里就可以设置target index的primary shard的数量。

curl -XPOST 'http://elasticsearch02:9200/twitter/\_shrink/twitter\_shrinked?pretty' -d '

{

"settings": {

"index.number\_of\_replicas": 1,

"index.number\_of\_shards": 1,

"index.codec": "best\_compression"

}

}'

当然也是需要监控整个shrink的过程的，用GET \_cat/recovery?v即可。

6、rollover index

rollover命令可以将一个alias重置到一个新的索引上去，如果已经存在的index被认为太大或者数据太旧了。这个命令可以接收一个alias名称，还有一系列的condition。如果索引满足了condition，那么就会创建一个新的index，同时alias会指向那个新的index。比如下面的命令。举例来说，有一个logs-0000001索引，给了一个别名是logs\_write，接着发起了一个rollover的命令，如果logs\_write别名之前指向的那个index，也就是logs-0000001，创建了超过7天，或者里面的document已经超过了1000个了，然后就会创建一个logs-000002的索引，同时logs\_write别名会指向新的索引。

这个命令其实是很有用的，特别是针对这种用户访问行为日志的数据，或者是一些联机事务系统的数据的进入，你可以写一个shell脚本，每天0:00的时候就执行以下rollover命令，此时就判断，如果说之前的索引已经存在了超过1天了，那么此时就创建一个新的索引出来，同时将别名指向新的索引。自动去滚动创建新的索引，保持每个索引就只有一个小时，一天，七天，三天，一周，一个月。

类似用es来做日志平台，就可能分布式电商平台，可能订单系统的日志，单独的一个索引，要求的是保留最近3天的日志就可以了。交易系统的日志，是单独的一个索引，要求的是保留最近30天的日志。

curl -XPUT 'http://elasticsearch02:9200/logs-000001?pretty' -d '

{

"aliases": {

"logs\_write": {}

}

}'

# Add > 1000 documents to logs-000001

curl -XPUT 'http://elasticsearch02:9200/logs-000001/data/1?pretty' -d '

{

"userid": 1,

"page": 1

}'

curl -XPUT 'http://elasticsearch02:9200/logs-000001/data/2?pretty' -d '

{

"userid": 2,

"page": 2

}'

curl -XPUT 'http://elasticsearch02:9200/logs-000001/data/3?pretty' -d '

{

"userid": 3,

"page": 3

}'

curl -XPOST 'http://elasticsearch02:9200/logs\_write/\_rollover?pretty' -d '

{

"conditions": {

"max\_age": "1d",

"max\_docs": 3

}

}'

{

"acknowledged": true,

"shards\_acknowledged": true,

"old\_index": "logs-000001",

"new\_index": "logs-000002",

"rolled\_over": true,

"dry\_run": false,

"conditions": {

"[max\_age: 7d]": false,

"[max\_docs: 1000]": true

}

}

这个过程常见于网站用户行为日志数据，比如按天来自动切分索引，写个脚本定时去执行rollover，就会自动不断创建新的索引，但是别名永远是一个，对于外部的使用者来说，用的都是最新数据的索引。

举一个简单的例子，这块是怎么玩儿的，比如说用es做网站的实时用户行为分析，要求的是一个索引只要保留当日的数据就可以了，那么就可以用这个rollover的策略，确保每个索引都是包含当日的最新数据的。老的数据，就变成别的索引了，此时可以写一个shell脚本，删除旧的数据，这样的话，es里就保留当前最新的数据就可以了。也可以根据你的需求，就保留最近7天的数据，但是最新一天的数据在一个索引中，供分析查询使用。

默认情况下，如果已经存在的那个索引是用-符号加上一个数字结尾的，比如说logs-000001，那么新索引的名称就会是自动给那个数字加1，比如logs-000002，自动就是给一个6位的数字，而且会自动补零。但是我们也可以自己指定要的新的索引名称，比如下面这样：

POST /my\_alias/\_rollover/my\_new\_index\_name

{

"conditions": {

"max\_age": "7d",

"max\_docs": 1000

}

}

可以将rollover命令和date日期结合起来使用，比如下面的例子，先创建了一个logs-2016.10.31-1格式的索引。接着每次如果成功rollover了，那么如果是在当天rollover了多次，那就是当天的日期，末尾的数字递增。如果是隔天才rollover，会自动变更日期，同时维护末尾的数字序号。

PUT /%3Clogs-%7Bnow%2Fd%7D-1%3E

{

"aliases": {

"logs\_write": {}

}

}

PUT logs\_write/log/1

{

"message": "a dummy log"

}

POST logs\_write/\_refresh

# Wait for a day to pass

POST /logs\_write/\_rollover

{

"conditions": {

"max\_docs": "1"

}

}

当然，还可以在rollover的时候，给新的index进行新的设置：

POST /logs\_write/\_rollover

{

"conditions" : {

"max\_age": "7d",

"max\_docs": 1000

},

"settings": {

"index.number\_of\_shards": 2

}

}

## Elasticsearch 生产集群中的索引管理（二）

1、mapping管理

put mapping命令可以让我们给一个已有的索引添加一个新的type，或者修改一个type，比如给某个type加一些字段

下面这个命令是在创建索引的时候，直接跟着创建一个type

curl -XPUT 'http://elasticsearch02:9200/twitter?pretty' -d '

{

"mappings": {

"tweet": {

"properties": {

"message": {

"type": "text"

}

}

}

}

}'

下面这个命令是给一个已有的索引添加一个type

curl -XPUT 'http://elasticsearch02:9200/twitter/\_mapping/user?pretty' -d '

{

"properties": {

"name": {

"type": "text"

}

}

}'

下面这个命令是给一个已有的type添加一个field

curl -XPUT 'http://elasticsearch02:9200/twitter/\_mapping/tweet?pretty' -d '

{

"properties": {

"user\_name": {

"type": "text"

}

}

}'

curl -XGET 'http://elasticsearch02:9200/twitter/\_mapping/tweet?pretty'，上面这行命令可以查看某个type的mapping映射信息

curl -XGET 'http://elasticsearch02:9200/twitter/\_mapping/tweet/field/message?pretty'，这行命令可以看某个type的某个field的映射信息

mapping管理是运维中，索引管理中，很基础的一块

2、索引别名管理

curl -XPOST 'http://elasticsearch02:9200/\_aliases?pretty' -d '

{

"actions" : [

{ "add" : { "index" : "twitter", "alias" : "twitter\_prod" } }

]

}'

curl -XPOST 'http://elasticsearch02:9200/\_aliases?pretty' -d '

{

"actions" : [

{ "remove" : { "index" : "twitter", "alias" : "twitter\_prod" } }

]

}'

POST /\_aliases

{

"actions" : [

{ "remove" : { "index" : "test1", "alias" : "alias1" } },

{ "add" : { "index" : "test2", "alias" : "alias1" } }

]

}

POST /\_aliases

{

"actions" : [

{ "add" : { "indices" : ["test1", "test2"], "alias" : "alias1" } }

]

}

上面是给某个index添加和删除alias的命令，还有重命名alias的命令（先删除再添加），包括将一个alias绑定多个index

POST /\_aliases

{

"actions" : [

{

"add" : {

"index" : "test1",

"alias" : "alias2",

"filter" : { "term" : { "user" : "kimchy" } }

}

}

]

}

DELETE /logs\_20162801/\_alias/current\_day

GET /\_alias/2016

索引别名，还是挺有用的，主要是什么呢，就是说，可以将一个索引别名底层挂载多个索引，比如说7天的数据

索引别名常常和之前讲解的那个rollover结合起来，我们为了性能和管理方便，每天的数据都rollover出来一个索引，但是在对数据分析的时候，可能是这样子的，有一个索引access-log，指向了当日最新的数据，用来计算实时数据的; 有一个索引access-log-7days，指向了7天的7个索引，可以让我们进行一些周数据的统计和分析。

3、index settings管理

curl -XPUT 'http://elasticsearch02:9200/twitter/\_settings?pretty' -d '

{

"index" : {

"number\_of\_replicas" : 1

}

}'

curl -XGET 'http://elasticsearch02:9200/twitter/\_settings?pretty'

经常可能要对index做一些settings的调整，常常和之前的index open和close结合起来

4、index template管理

我们可以定义一些index template，这样template会自动应用到新创建的索引上去。template中可以包含settings和mappings，还可以包含一个pattern，决定了template会被应用到哪些index上。而且template仅仅在index创建的时候会被应用，修改template，是不会对已有的index产生影响的。

curl -XPUT 'http://elasticsearch02:9200/\_template/template\_access\_log?pretty' -d '

{

"template": "access-log-\*",

"settings": {

"number\_of\_shards": 2

},

"mappings": {

"log": {

"\_source": {

"enabled": false

},

"properties": {

"host\_name": {

"type": "keyword"

},

"created\_at": {

"type": "date",

"format": "EEE MMM dd HH:mm:ss Z YYYY"

}

}

}

},

"aliases" : {

"access-log" : {}

}

}'

curl -XDELETE 'http://elasticsearch02:9200/\_template/template\_access\_log?pretty'

curl -XGET 'http://elasticsearch02:9200/\_template/template\_access\_log?pretty'

curl -XPUT 'http://elasticsearch02:9200/access-log-01?pretty'

curl -XGET 'http://elasticsearch02:9200/access-log-01?pretty'

index template，可能是这样子的，就是你可能会经常创建不同的索引，比如说商品，分成了多种，每个商品种类的数据都很大，可能就是说，一个商品种类一个索引，但是每个商品索引的设置是差不多的，所以干脆可以搞一个商品索引模板，然后每次新建一个商品种类索引，直接绑定到模板，引用相关的设置

## Elasticsearch 生产集群中的索引管理（三）

1、indice stat

indice stat对index上发生的不同类型的操作都提供了统计。这个api提供了index level的统计信息，不过大多数统计信息也可以从node level获取。

curl -XGET 'http://elasticsearch02:9200/twitter/\_stats?pretty'

这里包括了doc数量，index size，segment的内存使用量，merge，flush，refresh，translog等底层机制的统计信息。

2、segment

查看low level的lucene的segment信息，可以用来查看shard和index的更多的信息，包括一些优化信息，因为delete而浪费的数据空间，等等。

curl -XGET 'http://elasticsearch02:9200/twitter/\_segments?pretty'

{

...

"\_3": {

"generation": 3,

"num\_docs": 1121,

"deleted\_docs": 53,

"size\_in\_bytes": 228288,

"memory\_in\_bytes": 3211,

"committed": true,

"search": true,

"version": "4.6",

"compound": true

}

...

}

\_3，是segment的名称，这个名称跟这个segment files的文件名有关系，一个segment的所有文件都是用这个名称开头的

generation：每次新生成一个segment，就会递增一个数值，segment名称也就是这个数值

num\_docs：在这个segment中存储的没有被删除的document的数量

deleted\_docs：在这个segment中存储的被删除的document数量，这个数值是无所谓的，因为每次segment merge的时候都会删除这些document

size\_in\_bytes：这个segment占用的磁盘空间

memory\_in\_bytes：segment需要将一些数据缓存在内存中，这样搜索性能才能更高，这个数值就是segment占用的内存的空间大小

committed：segment是否被sync到磁盘上去了，commit/sync的segment可以确保数据不会丢失，但是即使这个值是false也不要紧，因为数据同时被存储在了translog里面，es进程重启的时候，是可以重放translog中的日志来恢复数据的

search：这个segment能不被搜索，如果是false的话，可能这个segment已经被sync到磁盘上，但是还没有进行refresh，所以不能被搜索

version：lucene的版本号

compound：如果是true的话，意味着lucene将这个segment所有的文件都merge成了一个文件，进而可以节省file descriptor的消耗

3、shard存储信息

查询索引shard copy的存储信息，可以看到哪些节点上有哪些shard copy，shard copy的allocation id，每个shard copy的唯一标识，包括打开索引的时候遇到的报错。默认情况下，会显示至少有一个未分配的copy的shard，如果cluster health是yellow，会显示至少有一个未分配的replica的shard，当cluster health是red，会显示有未分配的primary的shard。但是用status=green可以看到每个shard的信息。

curl -XGET 'http://elasticsearch02:9200/twitter/\_shard\_stores?pretty'

curl -XGET 'http://elasticsearch02:9200/twitter/\_shard\_stores?status=green&pretty'

{

...

"0": {

"stores": [

{

"sPa3OgxLSYGvQ4oPs-Tajw": {

"name": "node\_t0",

"transport\_address": "local[1]",

"attributes": {

"mode": "local"

}

},

"allocation\_id": "2iNySv\_OQVePRX-yaRH\_lQ",

"legacy\_version": 42,

"allocation" : "primary" | "replica" | "unused",

"store\_exception": ...

},

...

]

},

...

}

0：shard id

stores：shard的每个copy的store信息

sPa3OgxLSYGvQ4oPs-Tajw：node id，持有一个copy的node信息

allocationi\_id：copy的allocationid

allocation：shard copy的角色

4、clear cache

curl -XPOST 'http://elasticsearch02:9200/twitter/\_cache/clear?pretty'，这个命令可以清空所有的缓存

5、flush

flush API可以让我们去强制flush多个索引，索引flush以后，就会释放掉这个索引占用的内存，因为会将os cache里的数据强制fsync到磁盘上去，同时还会清理掉translog中的日志。默认情况下，es会不定时自动触发flush操作，以便于及时清理掉内存。POST twitter/\_flush，这条命令即可。

flush命令可以接受下面两个参数，wait\_if\_going，如果设置为true，那么flush api会等到flush操作执行完以后再返回，即使需要等待其他的flush操作先完成。默认的值是false，这样的话，如果有其他flush操作在执行，就会报错；force，如果没有必要flush的话，是不是会强制一个flush

curl -XPOST 'http://elasticsearch02:9200/twitter/\_flush?pretty'

6、refresh

refresh用来显式的刷新一个index，这样可以让这个refresh之前执行的所有操作，都处于可见的状态。POST twitter/\_refresh

curl -XPOST 'http://elasticsearch02:9200/twitter/\_refresh?pretty'

7、force merge

force merge API可以强制合并多个索引文件，可以将一个shard对应的lucene index的多个segment file都合并起来，可以减少segment file的数量。POST /twitter/\_forcemerge。

curl -XPOST 'http://elasticsearch02:9200/twitter/\_forcemerge?pretty'

## Elasticsearch 生产集群中的索引管理（四）

1、circuit breaker

es有很多的断路器，也就是circuit breaker，可以用来阻止各种操作导致OOM内存溢出。每个断路器都有一个限制，就是最多可以使用多少内存。此外，还有一个父断路器指定了所有断路器最多可以使用多少内存。

（1）父短路器

indices.breaker.total.limit，可以配置父短路器的最大内存限制，默认是jvm heap内存的70%

（2）fielddata短路器

field data短路器可以估算每一个field的所有数据被加载到内存中，需要耗费多大的内存。这个短路器可以组织field data加载到jvm内存时发生OOM问题。默认的值是jvm heap的60%。indices.breaker.fielddata.limit，可以用这个参数来配置。indices.breaker.fielddata.overhead，可以配置估算因子，估算出来的值会乘以这个估算因子，留一些buffer，默认是1.03。

（3）request circuit breaker

request circuit breaker会阻止每个请求对应的一些数据结构造成OOM，比如一个聚合请求可能会用jvm内存来做一些汇总计算。indices.breaker.request.limit，最大是jvm heap的60%。indices.breaker.request.overhead，估算因子，默认是1.

（4）in flight request circuit breaker

flight request circuit breaker可以限制当前所有进来的transport或http层的请求超出一个节点的内存总量，这个内存的使用量就是请求自己本身的长度。network.breaker.inflight\_requests.limit，默认是jvm heap的100%。network.breaker.inflight\_requests.overhead，估算因子，默认是1.

（5）script compilation circuit breaker

这个短路器可以阻止一段时间内的inline script编译的数量。script.max\_compilations\_per\_minute，默认是1分钟编译15个。

2、fielddata

fielddata cache，在对field进行排序或者聚合的时候，会用到这个cache。这个cache会将所有的field value加载到内存里来，这样可以加速排序或者聚合的性能。但是每个field的field data cache的构建是很成本很高昂的，因此建议给机器提供充足的内存来保持fielddata cache。

indices.fielddata.cache.size，这个参数可以控制这个cache的大小，可以是30%这种相对大小，或者是12GB这种绝对大小，默认是没有限制的。

fielddata的原理之前讲解过了，其实是对分词后的field进行排序或者聚合的时候，才会使用fielddata这种jvm内存数据结构。如果是对普通的未分词的field进行排序或者聚合，其实默认是用的doc value数据结构，是在os cache中缓存的。

3、node query cache

query cache用来缓存query的结果，每个node都有一个query cache，使用的是LRU策略，会自动清理数据。但是query cache仅仅会对那些filter后的数据进行缓存，对search后的数据是不会进行缓存的。indices.queries.cache.size，控制query cache的大小，默认是jvm heap的10%。

如果只是要根据一些field进行等值的查询或过滤，那么用filter操作，性能会比较好，query cache

4、index buffer

index buffer用来存储最新索引的的document。如果这个buffer满了之后，document就会被写入一个segment file，但是此时其实是写入os cache中，没有用fsync同步到磁盘，这就是refresh过程，写入os cache中，就可以被搜索到了。然后flush之后，就fsync到了磁盘上。indices.memory.index\_buffer\_size，控制index buffer的大小，默认是10%。indices.memory.min\_index\_buffer\_size，buffer的最小大小，默认是48mb。

index buffer，增删改document，数据先写入index buffer，写到磁盘文件里面去，不可见的，refresh刷入磁盘文件对应的os cache里面，还有translog一份数据

5、shard request cache

对于分布式的搜索请求，相关的shard都会去执行搜索操作，然后返回一份结果集给一个coordinate node，由那个coordinate node来执行最终的结果合并与计算。shard request cache会缓存每个shard的local result。那么对于频繁请求的数据，就可以直接从cache中获取了。与query cache不同的是，query cache只是针对filter的，但是shard request cache是针对所有search和聚合求的。

默认情况下，shard request cache仅仅会针对size=0的搜索来进行缓存，仅仅会缓存hits.total，聚合结果等等汇总结果，而不会缓存搜索出来的hits明细数据。

cache是很智能的，如果cache对应的doc数据被refresh，也就是修改了，那么cache就会自动失效。如果cache满了的话，也会自动用LRU算法来清理掉cache中的数据。

可以手动来启用和禁用cache：

PUT /my\_index

{

"settings": {

"index.requests.cache.enable": false

}

}

在每个request中也可以手动启用或禁用cache：

GET /my\_index/\_search?request\_cache=true

{

"size": 0,

"aggs": {

"popular\_colors": {

"terms": {

"field": "colors"

}

}

}

}

但是默认对于size>0的request的结果是不会被cache的，即使在index设置中启用了request cache也不行。只有在请求的时候，手动加入reqeust cache参数，才可以对size>0的请求进行result cache。

缓存用的key，是完整的请求json，因此每次请求即使json中改变了一点点，那么也无法复用上次请求的request cache结果。

indices.requests.cache.size，可以设置request cache大小，默认是1%

GET /\_stats/request\_cache?human，监控request cache的使用

如果是search，默认是不缓存的，除非你手动打开request\_cache=true，在发送请求的时候

如果是aggr，默认是缓存的，不手动打开request\_cache=true，也会缓存聚合的结果

6、索引恢复

indices.recovery.max\_bytes\_per\_sec，每秒可以恢复的数据量，默认是40mb

## Elasticsearch 生产集群中的索引管理（五）

1、merge

es里的一个shard，就是一个lucene index，每个lucene index都是由多个segment file组成的。segment file负责存储所有的document数据，而且segment file是不可变的。一些小的segment file会被merge成一个大的segment file，这样可以保证segment file数量不会膨胀太多，而且可以将删除的数据实际上做物理删除。merge过程会自动进行throttle限流，这样可以让merge操作和节点上其他的操作都均衡使用硬件资源。

merge scheduler会控制merge的操作什么时候执行，merge操作会在一些独立的后台线程中执行，如果达到了最大的线程数量的话，那么merg操作就会等待有空闲的线程出来再去执行。index.merge.scheduler.max\_thread\_count，这个参数可以控制每次merge操作的最大线程数量，默认的公式是Math.max(1, Math.min(4, Runtime.getRuntime().availableProcessors() / 2))，对于SSD来说，表现是很好的。但是如果我们使用的是机械硬盘，建议将这个数量降低为1。

2、translog

（1）translog介绍

对lucene的磁盘持久化，可以通过一次lucene commit来提交，是一个很重的过程，所以是不可能在每次插入或者删除操作过后来执行的。所以在上一次提交和下一次提交之间的数据变更，都是在os cache中的，如果机器挂掉，可能导致数据丢失。为了避免这种数据丢失，每个shard都有一个transaction log，也就是translog，来写入write ahead log，预写日志。任何写入数据都会同时被写入translog。如果机器挂掉了，那么就可以重放translog中的日志来恢复shard中的数据。

一次es flush操作，都会执行一次lucene commit，将数据fsync到磁盘上去，同时清空translog文件。在后台会自动进行这个操作，来确保translog日志不会增长的太过于巨大，这样的话重放translog数据来恢复，才不会太慢。index.translog.flush\_threshold\_size，这个参数可以设置os cache中数据达到多大的时候，要执行一次flush，默认是512mb。

（2）translog设置

此外，默认情况下，es每隔5秒钟会在每个增删改操作结束之后，对translog进行一次fsync操作，但是要求index.translog.durability设置为async或者request（默认）。es只有在primary和每一个replica shard的translog fsync之后，才会对增删改操作返回的状态中显示为success。

index.translog.sync\_interval：不考虑些操作，translog被fsync到磁盘的频率，默认是5秒

index.translog.durability：是否要在每次增删改操作之后，fsync translog。

默认是request，每次写请求之后，都会fsync translog，这样的话，即使机器宕机，但是只要是返回success的操作，就意味着translog被fsync到磁盘了，就可以保证数据不丢失

async，在后台每隔一定时间来fsync translog，默认是5秒，机器宕机的时候，可能导致5秒的数据丢失，可能有5秒钟的数据，数据本身是停留在os cache中的，数据对应的translog也停留在os cache中，5秒才能将translog从os cache输入磁盘

（3）translog损坏怎么办

如果硬盘损坏，可能导致translog出现损坏，es会自动通过checksum来捕获到这个问题，此时es就会认为这个shard故障了，而且禁止将shard分配给这个node，同时会尝试从其他replica shard来恢复数据。如果没有replica数据的话，那么用户可以手动通过工具来恢复，使用elasticsearch-translog即可。要注意的是，elasticsaerch-translog工具不能再elasticsearch运行的时候来使用，否则我们可能丢失数据。

bin/elasticsearch-translog truncate -d /var/lib/elasticsearchdata/nodes/0/indices/P45vf\_YQRhqjfwLMUvSqDw/0/translog/

Checking existing translog files

## Elasticsearch 底层模块深入解析-shard allocation

跟大家解释一下，大家可能会发现，最近几讲，好像有一点点务虚，因为很少实操，大量的是解释和讲解

主要还是运维相关的底层知识，为什么说实操比较少，涉及到的很多参数，不太好设置，需要做到一些知识储备

1、shard allocation的介绍

两种node：master node，data node

master node的一个很重要的功能，比如说，你现在创建了一个索引，然后这个索引是不是有很多的shard，可能你自己指定了几个primary shard，每个primary shard还有一些replica shard。master node，其实就是决定哪些shard分配给哪些node，以及什么时候在node之间移动shard，来让集群达到rebalance。对于shard allocation而言，有很多设置，都可以控制这个过程：

（1）cluster level shard allocation，可以在集群层面来控制shard allocation和rebalance的过程

（2）disk-based shard allocation，es会在shard分配的时候，考虑可用的磁盘空间

（3）shard allocation awareness，控制shard如何在不同的机架上进行分布

（4）shard allocation filter，可以控制有些node不参与allocation的过程，这样的话，这些node就可以被安全的下线

2、cluster level shard allocation

shard allocation，就是将shard分配给node的一个过程，这个过程可能会在集群启动初始化进行恢复的时候发生，也会发生在replica shard被分配的时候，集群进行rebalance的时候，或者是有新的node加入，有旧的node被下线的时候。

（1）shard allocation settings

cluster.routing.allocation.enable

all，默认，对所有类型的shard都允许分配

primaries，仅仅允许primary shard被分配

new\_primaries，仅仅对新建索引的primary shard允许分配

none，不允许任何shard被分配

但是这个配置对node重启时本地primary shard的恢复没有影响，重启node的时候，如果本地有一个未被分配的primary shard，还是会立即恢复这个primary shard。

cluster.routing.allocation.node\_concurrent\_incoming\_recoveries：在一个node上允许同时恢复多少个shard，这里的shard recovery过程，指的就是要将某个shard分配给这个node。这个设置的默认值是2.

cluster.routing.allocation.node\_concurrent\_outgoing\_recoveries：一个node上允许同时进行多少个shard recovery outgoing，比如这个node上，有一个primary shard，现在要将replica shard分配给其他的node，那么就是outgoing shard recovery。默认值也是2.

cluster.routing.allocation.node\_concurrent\_recoveries：同时设置上面两个值

cluster.routing.allocation.node\_initial\_primaries\_recoveries：如果replica shard recovery通过网络传输来分配，那么一个未被分配的primary shard会在node重启之后使用本地磁盘上的数据，这个过程因为是使用本地的数据，因此会比较快，默认值是4.

\*\*\*\*\*：cluster.routing.allocation.same\_shard.host：默认值是false，如果设置为true，那么就不允许将一个primary shard和replica shard分配到同一个物理机上，也许这个物理机上启动了多个es实例。

有可能你有一台超级服务器，32核CPU+128G内存，这个时候的话呢，可能你在这台机器上启动两个es进程，但是默认情况下，有可能一个shard的primary shard被分配到了这台物理机上的node1，同时这个primary shard的replica shard被分配到了这台物理机上的node2，此时，primary shard和replica shard就在同一台物理机上了。

可用性是比较低的，因为如果这台物理机挂掉了，比较惨烈了，primary shard和replica shard全部丢失

（2）shard rebalance settings

rebalance，什么意思，比如说你的集群，有5台机器，一共有100个shard，负载均衡的情况下，平均分配一下呢，每个机器上有20个shard。然后此时加入了一台新机器，6台机器了，此时就要触发rebalance操作，重新让整个集群负载均衡，100 / 6 = 16~17个shard每台机器

如果下线一台机器的话。。。

cluster.routing.rebalance.enable

all，默认，允许对所有类型的shard进行rebalance过程

primaries，仅仅允许对primary shard进行rebalance过程

replicas，仅仅允许对replica shard进行rebalance

none，不允许对任何shard进行rebalance

cluster.routing.allocation.allow\_rebalance

always，任何时候都允许rebalance

indices\_primaries\_active，仅仅只有在所有的primary shard都被分配之后才允许rebalance

indices\_all\_active，默认，仅仅允许所有的primary shard和replica shard都被分配之后，才能rebalance

cluster.routing.allocation.cluster\_concurrent\_rebalance

允许控制多少个shard rebalance的操作同时运行，默认是2

（3）shard balance heuristics

cluster.routing.allocation.balance.shard：设置每个node的shard分配的权重因子，默认是0.45f，提高权重因子，就会尽可能让均匀的shard分配给集群中的所有node

cluster.routing.allocation.balance.index：定义每个index在一个node上的shard数量因子，默认是0.55f，提高这个参数，就会尽可能让每个index的shard均匀分配到所有的node上

cluster.routing.allocation.balance.threshold：默认是1.0f，提高这个权重因子会导致集群对shard balance有更小的侵略性

3、disk-based shard allocation

es在进行shard allocation的时候，会充分考虑每一个node的可用磁盘空间

cluster.routing.allocation.disk.threshold\_enabled：默认是true，如果是false会禁用基于disk的考虑

cluster.routing.allocation.disk.watermark.low：控制磁盘使用率的低水位，默认是85%，如果一个节点的磁盘空间使用率已经超过了85%，那么就不会分配shard给这个node了

cluster.routing.allocation.disk.watermark.high：控制磁盘使用率的高水位，默认是90%，如果一个节点的磁盘空间使用率已经超过90%了，那么就会将这个node上的部分shard移动走

cluster.info.update.interval：es检查集群中每个node的磁盘使用率的时间间隔，默认是30s

cluster.routing.allocation.disk.include\_relocations：默认是true，意味着es在计算一个node的磁盘使用率的时候，会考虑正在分配给这个node的shard。

4、shard allocation awareness

（1）机架感知特性

如果在一个物理机上运行多个虚拟机，并且在多个虚拟机中运行了多个es节点，或者在多个机架上，多个机房，都有可能有多个es节点在相同的物理机上，或者在相同的机架上，或者在相同的机房里，那么这些节点就可能会因为物理机，机架，机房的问题，一起崩溃

如果es可以感知到硬件的物理布局，就可以确保说，priamry shard和replica shard一定是分配到不同的物理机，或者物理机架，或者不同的机房，这样可以最小化物理机，机架，机房崩溃的风险

shard allocation awareness可以告诉es我们的硬件架构

举哥例子，如果我们有多个机架，那么我们启动一个node的时候，就要告诉这个node它在哪个机架上，可以给它一个rack\_id，比如下面的命令：./bin/elasticsearch -Enode.attr.rack\_id=rack\_one，也可以在elasticsearch.yml中设置这个机架id

cluster.routing.allocation.awareness.attributes: rack\_id

node.attr.rack\_id=rack\_one

上面的两行设置里，第一行是设置机架id的属性名称，第二行是用那个机架id属性名称设置具体的机架id

如果启动两个node，都在一个机架上，此时创建一个有5个primary shard和5个replica shard的索引，此时shards会被分配到两个节点上

如果再启动两个node，设置为另外一个机架，此时es会将shard移动到新的node上，去确保说，不会让primary shard和其replica shard在同一个机架上。但是如果机架2故障了，为了恢复集群，那么还是会在恢复的时候，将shards全部在机架1上分配的

prefer local shard机制：在执行search或者get请求的时候，如果启用了shard awareness特性，那么es会尽量使用local shard来执行请求，也就是在同一个awareness group中的shard来执行请求，也就是说尽量用一个机架或者一个机房中的shard来执行请求，而不要跨机架或者跨机房来执行请求

可以指定多个awareness属性，比如机架id和机房名称，类似下面：cluster.routing.allocation.awareness.attributes: rack\_id,zone

（2）强制性的感知

如果现在我们有两个机房，并且有足够的硬件资源来容纳所有的shard，但是可能每个机房的硬件只能容纳一半shard，不能容纳所有的shard。如果仅仅使用原始的感知特性，如果一个机房故障了，那么es会将需要恢复的shard全部分配给剩下的一个机房，但是剩下的那个机房的硬件资源并不足以容纳所有的shard。

强制感知特性会解决这个问题，因为这个特性会绝对不允许在一个机房内分配所有的shard

比如说，有一个感知属性叫做zone，有两个机房，zone1和zone2，看看下面的配置：

cluster.routing.allocation.awareness.attributes: zone

cluster.routing.allocation.awareness.force.zone.values: zone1,zone2

那么此时如果将2个node分配给zone1机房，然后创建一个索引，5个primary shard和5个replica shard，但是此时只会在zone1机房分配5个primary shard，只有我们启动一批node在zone2机房，才会分配replica shard

5、shard allocation filtering

shard allocation filtering可以让我们允许或者不允许某些index的shard分配给一些特殊的节点，典型的用途，就是如果我们要下线一些node，就可以用这个feature禁止shard分配给这些即将下线的node，而且我们还可以将这些即将下线的节点的shard移动到其他节点。

用下面的命令可以下线一个节点，因为就不允许将shard分配给这个节点了

PUT \_cluster/settings

{

"transient" : {

"cluster.routing.allocation.exclude.\_ip" : "10.0.0.1"

}

}

6、node下线时的shard延迟分配

如果从集群中下线一个节点，master会做下面这些事情：

（1）如果那个节点上有primary shard，那么master会将那些primary shard在其他节点上的replica shard提升为primary shard

（2）分配新的replica shard来保证replica数量充足

（3）在剩下的各个node上进行shard rebalance，确保负载均衡

这些操作可以保护集群不会丢失数据，因为会对每个shard都复制充足的replica shard

但是这个过程，可能会导致集群中出现很重的负载，包括网络负载和磁盘IO负载，如果那个下线的节点只是因为故障被下线，马上就会有新的节点来顶替它，那么这种立即执行的shard recovery过程是不需要的，考虑下面的场景：

（1）某个node跟集群丢失了网络连接

（2）master node将那个node上的primary shard对应的其他节点上的replica shard提升为primary shard

（3）master node分配新的replica shard到其他节点上

（4）每个新的replica shard都会通过网络传输一份primary shard的完整的副本数据

（5）很多shard都被移动到其他的node来让集群rebalance

（6）但是几分钟以后，那个丢失了网络连接的node又重新连接到了集群中

（7）master节点又要再次进行rebalance操作，因为需要将一些shard分配给那个node

其实如果master node也许只要等待几分钟，那么丢失的那个node自己会回来，丢失的shard也会自动恢复过来，因为数据都在节点的本地，不需要重新拷贝数据以及网络传输，这个过程是非常快速的

index.unassigned.node\_left.delayed\_timeout，这个参数可以设置某个节点下线之后，对应的replica shard被重新复制和分配的时间等待期，默认是1m，可以通过下面的命令来修改：

PUT \_all/\_settings

{

"settings": {

"index.unassigned.node\_left.delayed\_timeout": "5m"

}

}

如果启用了delayed allocation之后，那么就会看到下面的场景：

（1）某个node丢失了网络连接

（2）master将那个node上的一些primary shard对应的其他node上的replica shard提升为primary shard

（3）master记录下来一条消息日志，这个primary shard的replica shard还没有重新分配和开始，被delayed了，会等待1m

（4）cluster会保持yellow状态，因为没有足够的replica shard

（5）那个丢失了的node在几分钟之后，如果回到了集群中

（6）缺失的那些replica shard会直接分配给那个node，使用其本地的数据即可

如果某个node确定了肯定不会再回到集群中，那么可以通过下面的命令，手动设置一下，直接不等待那个节点回来了

PUT \_all/\_settings

{

"settings": {

"index.unassigned.node\_left.delayed\_timeout": "0"

}

}

7、索引恢复的优先级

没有被分配的shard都是按照优先级来分配的，有下面几个优先级，index.priority，索引的创建日期，索引名称

PUT index\_3

{

"settings": {

"index.priority": 10

}

}

8、每个节点的shard数量

cluster.routing.allocation.total\_shards\_per\_node，设置每个节点最多承载的shard数量，默认是无限制的

## Elasticsearch 底层模块深入解析-gateway

gateway，elasticsearch底层的一个module，这个module，你也可以认为是什么呢？认为是es代码中的一个模块

gateway这个模块是负责干什么的，module，java，j2ee，java web，用户管理模块，订单管理模块。。。用户管理模块，就是类似一个module，是用来管理用户信息的

elasticsearch底层模块，英文，module，类似用户管理模块，订单管理模块，gateway module，是用来进行es自身的一些元数据，比如说cluster state，里面包含了一些集群中有哪些node，每个node的信息，负载，资源，索引，每个索引的shard在各个node上分配的一些信息啊。。。

gateway module，是负责用来存储每个es节点的cluster state的，node重启的时候，gateway也会负责将本地磁盘上的cluster state给它读取出来，放入内存中

local gateway module，用来存储cluster state，并且在集群重启的时候共享数据

以下的设置，必须在每个master候选节点上都进行设置

gateway.expected\_nodes：要求必须有多少个节点在集群中，当加入集群中的节点数量达到这个期望数值之后，每个node的local shard的恢复就会理解开始，默认的值是0，也就是不会做任何的等待

gateway.expected\_master\_nodes：要求必须有多少个master node在集群中，只要有这个数量的master node加入了集群，每个node的local shard recovery就会立即开始，默认的值是0

gateway.expected\_data\_nodes：要求必须有多少个data node在集群中，只要有这个数量的master node加入了集群，每个node的local shard recovery就会立即开始，默认的值是0

gateway.recover\_after\_time：如果期望的节点数量没有达标，那么会等待一定的时间，然后就开始进行shard recovery，默认是等待5m

如果gateway.recover\_after\_time时间范围内，指定数量的节点还没有加入集群，但是只要满足下面的条件之一就会立即开始恢复

gateway.recover\_after\_nodes：只要指定数量的节点加入集群，就开始进行恢复

gateway.recover\_after\_master\_nodes：只要指定数量的master节点加入集群，就开始进行恢复

gateway.recover\_after\_data\_nodes：只要指定数量的data node加入集群，就开始恢复

比如说，集群中一共有10个node

gateway.recover\_after\_nodes: 8

gateway.expected\_nodess: 10

gateway.recover\_after\_time: 10m

要求集群中达到8个节点，接下来，如果等待超过10分钟那么，就会开始shard recovery，或者是到了8个节点之后，在10分钟之内，立即到了10个节点，那么也会立即开始shard recovery

## Elasticsearch 底层模块深入解析-http,network,transport

来讲解这个elasticsearch所谓的这些底层Module，有一点点对之前的课程炒冷饭的嫌疑，不是我想要炒冷饭，而是说，之前主要是站在实际使用和运维等等角度去讲解这个es。但是之前讲解的很多功能和特性，参数，实际上都是围绕着es的一些module来的。系统梳理一下es的一些底层module

shard allocation module

gateway module

http module

network module

transport module

我们就讲解http，network，transport三个module最最常用的知识和参数

1、http module

HTTP module就是es的http api模块

http机制是完全异步的，也就是说线程不会因为等待响应而陷入阻塞，http异步通信机制的优点就是解决了C10k问题

如果可能的话，尽量使用http keep alive，可以提升性能，而且可以避免客户端发生http chunking现象

下面是一些常用的http设置

http module，主要是用来对外提供请求接口服务的，我们不是会向es发送一个rest请求，其实就是走es的http module的，其实就是用来处理外部的请求的

http.port，es对外暴露的http api接口的端口号，默认在9200~9300之间选择一个，优先选择9200，如果被绑定，则选择9201，以此类推

我们之前用curl工具，发送http请求，那么其实就是走es的http module，还是http.port设置的http module监听的端口号

默认的话，http.port就是9200，如果9200被占用，那么就会用9201，以此类推，一直到9300

2、network module

es默认是绑定到localhost的，这只能让es运行在开发模式下，如果要运行在生产模式下，下面的一些network设置是必须设置的

network.host：节点绑定的hostname或者ip地址，设置之后，才能进入生产模式下

主要是对一些网络上的基础性的东西进行一个配置

network.host，绑定的是本地的回环地址，127.0.0.1，进入的是development mode，开发模式

如果将network.host，设置为比如192.168.31.187之类的这种hostname或者ip地址之后，进入production mode，生产模式

3、transport module

transport是用来进行节点间的互相通信的模块

transport.tcp.port：用于配置节点间互相通信的端口号，默认是9300，范围在9300~9400之间，优先绑定9300，如果被占用，则用9301，以此类推

transport module，es各个node之间，其实也会进行频繁的通信，比如交换cluster state，reqeust transfer，比如插入一条document，路由之后，应该是到node3的shard2上去处理，但是请求可能发送到的是node1的shard0上，node1就要将这个document index的请求转发给node3，让node3上的shard2去处理这个请求

默认transport.tcp.port端口号是9300，如果被占用，那么就会用9301，一直到9400，以此类推

## Elasticsearch 底层模块深入解析-threadpool

threadpool module，也是有一些内容要讲解的哦，这个module主要是用来干什么的呢。。。

每个es节点内部多有很多个thread pool，不同的thread pool会处理不同的请求，thread pool module就是用来对各种各样的thread pool进行管理的

每种thread pool都是绑定了一个queue的，因为thread pool的大小是有限的，比如一个thread pool内部就是10个线程，那么此时如果10个线程都被耗尽了，在执行某项任务，此时新的请求要这个thread pool中的线程来处理，会怎么样？默认情况下，肯定就惨了，因为没有线程处理了，可能就会报错了。。。。

那么但是es的thread pool都是绑定了一个内存中的队列的，queue，如果thread pool满了之后，请求可以进这个queue里面来排队，等待线程池出现空的线程来处理queue中的请求，这样的话呢，就提供了一个buffer

不至于说，在业务高峰期，大量的报错，因为线程池可能满了，至少用queue缓冲一下，也许请求会因为在queue中等待，执行的慢了一些，但是至少是不会报错，可以执行的s

1、线程池

每个节点都有多个thread pool，这样可以提升多线程处理的能力，这些线程池大多数都有一个对应的queue与其绑定，可以允许线程池满的时候，让pending的请求在队列里排队，而不是将pending请求抛弃掉

generic thread pool：应付一些普通的操作，比如后台的node discovery，thread pool类型是scaling

index thread pool：用于进行index和delete操作，是fixed类型，大小为cpu core数量，queue大小是200，这个线程池的最大大小是cpu core数量 + 1

search thread pool：用于search操作，是fixed类型，大小是cpu core数量 \* 3 / 2 + 1，queue大小是1000

get thread pool：用于get操作，是fixed类型，大小是cpu core数量，队列大小是1000

bulk thread pool：用于bulk操作，是fixed类型，大小是cpu core数量，queue大小是200，最大的线程池大小是cpu core数量 + 1

snapshot thread pool：用于snapshot/restore操作，是scaling类型，每个线程存活时间为5m，最大数量是min(5, cpu core数量 / 2)

refresh thread pool：用于refresh操作，是scaling类型，存活时间为5m，最大数量是min(10, cpu core数量 / 2)

用下面的方式来修改thread pool

正好给大家新鲜出炉的一个案例，在我进行这个运维这块课程升级之前，就有之前的学员，在实际的项目中，碰到一个case，就是执行bulk操作的时候，说线程池不够了，建议增加queue的数量，bulk thread pool，做一下设置

在elasticsearch.yml配置文件中，按照下面的格式来进行配置

thread\_pool:

bulk:

size: 16

queue\_size: 1000

2、线程池类型

fixed类型线程池：线程数量是固定的，同时绑定一个queue用于存放pending request

scaling类型：这种线程池数量是可变的，根据负载来变化，最小是cpu core数量，最大是其公式定义，keep\_alive参数可以控制其线程空闲多长时间被释放

thread\_pool:

refresh:

core: 1

max: 8

keep\_alive: 2m

3、cpu core数量设置

在elasticsearch.yml配置文件中去设置的

processors: 2

通过上面的参数可以显示设置cpu core数量，意义有下面3点：

（1）如果在一台机器上运行了多个es节点，但是可能只想要让每个es节点使用部分cpu core，而不是物理机上的所有cpu core，就可以手动设置。比如一台物理机，上面的cpu core是16个，运行了两个es节点，此时就可以手动设置processors是8，就是让每个es节点仅仅使用8个cpu core

（2）默认cpu core的数量最大限制为32个，所以如果我们如果物理机超过了32个cpu core，那么可以手动设置。比如说你的物理机的cpu core是64个，但是此时es会去使用的cpu core可能也就32个，最大限制，此时就是要手动设置processors是64。

（3）有时候可能会捕获到错误的cpu core数量，此时需要手动设置

## Elasticsearch 底层模块深入解析-plugin

plugin module，主要就是用来负责这个插件的管理，安装插件，查看插件，删除插件，更新插件

在elasticsearch的新版中，演示过了，插件应该怎么玩儿，hdfs repository的一个插件，怎么安装

安装plugin：elasticsearch-plugin install [plugin\_name]

通过url安装plugin：elasticsearch-plugin install [url]

如果你的机器可能是不能直接连外网的，那么你可能需要将plugin插件下载下来一个包，在本地离线安装这个插件，其实在实际的运维中，这个比较常见

通过本地包离线安装plugin：elasticsearch-plugin install file:///path/to/plugin.zip

查看plugin list：elasticsearch-plugin list

删除plugin：elasticsearch-plugin remove [pluginname]，如果删除了一个plugin，必须重启节点才能生效

升级/更新plugin：先删除，再安装，然后重启节点

比如说，你现在升级了es节点的版本，然后就得对应着升级你的plugin，自动装的就是最新版的了，当然装好以后，要重启es node才能生效

es jdbc，es spring data，es sql

## Elasticsearch 底层模块深入解析-node

node module，主要是用来处理各种不同类型的节点的，es有哪些类型的node，另外就是对这些类型的node有些什么特殊的参数，对于一个较大的集群来说，如何去规划和配置各种各样的node

1、node类型

如果我们启动es的一个实例，那么就是启动了一个es node，一些es node就可以组成一个es集群。如果仅仅运行了一个es node，那么也有一个es集群，只是节点数量就是1。

集群中的每个node都可以处理http和transport请求，其中transport层是用来处理节点间的通信的，http层是用来处理外部的客户端rest请求的。

所有的node都知道集群中的其他node，并且可以将客户端的请求转发到适当的节点上去。

节点的类型包含以下几种：

（1）master-eligible node：master候选节点，将node.master设置为true（默认），代表这个node就是master的候选节点，可以被选举为master node，然后控制整个集群。

（2）data node：将node.data设置为true（默认），data node可以存储数据，同时处理这些数据相关的操作，比如CRUD操作，搜索操作，聚合操作，等等。

（3）ingest node：将node.ingest设置为true（默认），ingest node是用来对document写入索引文件之前进行预处理的。可以对每个document都执行一条ingest pipeline，在document写入索引文件之前，先对其数据进行处理和转化。但是如果要执行的ingest操作太过繁重，那么可以规划单独的一批ingest node出来，然后将node.master和node.data都设置为false即可。

（4）tribe node：tribe node可以通过tribe.\*相关参数来设置，它是一种特殊的coordinate node，可以连接到多个es集群上去，然后对多个集群执行搜索等操作。

（5）默认情况下，每个node的node.master，node.data，node.ingest都是true，都是master候选节点，也可以作为data node存储和操作数据，同时也可以作为ingest node对数据进行预处理。对于小于20个节点的小集群来说，这种架构是ok的，没问题的。但是如果对于大于20个物理机的集群来说，最好是单独规划出master node、data node和ingest node来。

（6）coordinate node

搜索和bulk等请求可能会涉及到多个节点上的不同shard里的数据，比如一个search请求，就需要两个阶段执行，首先第一个阶段就是一个coordinating node接收到这个客户端的search request。接着，coordinating node会将这个请求转发给存储相关数据的node，每个data node都会在自己本地执行这个请求操作，同时返回结果给coordinating node，接着coordinating node会将返回过来的所有的请求结果进行缩减和合并，合并为一个global结果。

每个node都是一个coordinating node。这就意味着如果一个node，将node.master，node.data，node.ingest全部设置为false，那么它就是一个纯粹的coordinating node，仅仅用于接收客户端的请求，同时进行请求的转发和合并。

如果真的是大集群的话，最好也是单独规划一批node出来，就作为coordinating node，然后让es client全部往这些node上去发送请求。

如果真的是一个大于20个节点的生产集群的话，建议将4种node，master node，data node，ingest node，cooridating node，全部分离开来

集群中有30台机器

master node：3个

ingest node：视具体情况而定，具体是看你的ingest预处理操作有多么的复杂，耗费多少资源，但是一般情况下来说，es ingest node用的比较少的，ingest node也可以不用单独规划一批出来

coordinate node：视具体情况而定，但是对于大集群来说，最好是单独拆几个节点出来，用于接收客户端的请求，3个节点。主要是看你的并发访问量有多大，比如说你的最大的QPS也就是10，或者是100，那么3个节点肯定够了。如果你的QPS是1000，或者是10000，那么可能就要规划，10个coordinate node，或者100个

data node：24个data node，data node肯定是分配的是最多的，主要用来存储数据，执行各种对数据的操作么，资源耗费肯定是最多的

2、master eligible node

（1）master-eligible node的介绍以及配置

master node负责轻量级的集群管理工作，比如创建和删除索引，追踪集群中的每个node，决定如何将shards分配给各个node。对于集群来说，有一个稳定的master node，是非常关键的。然后master-eligible node都有机会被选举为一个master node，同时master node必须有权限访问path.data指定的data目录，因为master node需要在data目录中存储cluster state。

对数据进行index和search操作，会耗费大量的cpu，内存，磁盘io，以及网络io，耗费的是每个node的资源。因此我们必须要确保master node是非常稳定的，而且是压力不大的，对于大集群来说，比较好的办法是划分出单独的master node和data node。如果不拆开的话，一个node又要是data node，要复杂存储数据，处理各种操作，同时又要负责管理集群，可能就会不稳定，出问题。

同时因为默认情况下，master node也能扮演coordinating node的角色，并且将search和index请求路由到对应的data node上去执行，最好是不要让master node来执行这些coordinate操作。因为msater node的稳定运行对于整个集群来说非常重要，比你利用master node资源来执行一些coordinate操作要重要的多。

如果要设置一个node为专门的master-eligible node，需要做如下的设置：

node.master: true

node.data: false

node.ingest: false

（2）通过minimum\_master\_nodes来避免脑裂问题

要预防数据的丢失，我们就必须设置discovery.zen.minimum\_master\_nodes参数为一个合理的值，这样的话，每个master-eligible node才知道至少需要多少个master-eligible node才能组成一个集群。

比如说，我们现在有一个集群，其中包含两个master-eligible nodes。然后一个网络故障发生了，这两个节点之间丢失了联络。每个节点都认为当前只有一个master-eligible node，就是它们自己。此时如果discovery.zen.minimum\_master\_nodes参数的默认值是1，那么每个node就可以让自己组成一个集群，选举自己为master node即可。结果就会导致出现了两个es集群，这就是脑裂现象。即使网络故障解决了，但是这两个master node是不可能重新组成一个集群了。除非某个master eligible node重启，然后自动加入另外一个集群，但是此时写入这个节点的数据就会彻底丢失。

那么如果现在我们有3个master-eligible node，同时将discovery.zen.minimum\_master\_nodes设置为2.如果网络故障发生了，此时一个网络分区有1个node，另外一个网络分区有2个node，只有一个node的那个网络分区，没法检测到足够数量的master-eligible node，那么此时它就不能选举一个master node出来组成一个新集群。但是有两个node的那个网络分区，它们会发现这里有足够数量的master-eligible node，那么就选举出一个新的master，然后组成一个集群。当网络故障解除之后，那个落单的node就会重新加入集群中。

discovery.zen.minimum\_master\_nodes，必须设置为master-eligible nodes的quorum，quorum的公式为：(master\_eligible\_nodes / 2) + 1。

换句话来说，如果有3个master-eligible nodes，那么那个参数就必须设置为(3 / 2) + 1 = 2，比如下面这样：

discovery.zen.minimum\_master\_nodes: 2

随着集群节点的上线和下限，这个参数都是要重新设置的，可以通过api来设置

PUT \_cluster/settings

{

"transient": {

"discovery.zen.minimum\_master\_nodes": 2

}

}

此时将master node和data node分离的好处就出来了，一般如果单独规划一个master nodes的话，只要规划固定的3个node是master-eligible node就可以了，那么data node无论上线和下限多少个，都无所谓的。

3、data node

data node负责存储shard的数据，也就是那些document。data node可以处理各种操作，比如CRUD，搜索，聚合。这些操作全都是很耗费IO，内存和cpu资源的。因此监控这些资源的使用是很重要的，同时如果资源过载了，那么就要添加更多的data node。

如果要设置一个专门的data node，需要做出如下的设置：

node.master: false

node.data: true

node.ingest: false

4、ingest node

nigest node可以执行预处理pipeline，包含了多个ingest processors。不同的ingest processor执行的操作类型是不同的，那么对资源的需求也是不同的，不过还是最好是规划一批单独的ingest node出来，不要跟master node和data node混合在一起。

如果要配置一个单独的ingest node：

node.master: false

node.data: false

node.ingest: true

search.remote.connect: false

5、cooridnating only node

如果我们规划了一批专门的master node，data node以及ingest node，那么此时还遗留下来了一种node，那就是coordinating node，这些node专门用来接收客户端的请求，同时对请求进行路由和转发，并对请求的结果进行合并。

coordinating only nodes对于大集群来说，可以使用专门的node来负载coordinate操作，而不是让coordinate操作的工作负载集中到master node和data node上去。coordinating node也会加入cluster，同时可以获取到完整的cluster state，它们主要是用cluster state中包含的node info来进行请求转发。

如果在一个集群中规划太多的coordinating node可能会加重整个集群的负担，因为被选举出来的master node必须要从所有的node上得到cluster state update的ack，如果coordinating nodes过多，那么可能会加重master node的负担。

如果要设置coordinating only node的话：

node.master: false

node.data: false

node.ingest: false

search.remote.connect: false

6、node data path设置

（1）path.data

每个data和master-eligible node都需要能够访问data目录，在那里存储了每个shard的数据，包括cluster state也存储在那里。path.data默认是指向$ES\_HOME/data目录的，但是在生产环境中，肯定是不能这样设置的，因为在升级es的时候，可能会导致数据被清空或者覆盖。

此时一般需要在elasticsearch.yml中设置path.data：

path.data: /var/elasticsearch/data

（2）node.max\_local\_storage\_nodes

data目录可以被多个node共享，即使是不同集群中的es node，也许他们在一个物理机上启动了。这个共享的方式对于我们测试failover是很有用的，以及在开发机上测试不同的配置。但是在生产环境下，绝对不用这么做，一个data目录就给一个es node使用即可。默认情况下，es被配置成阻止超过一个node共享data目录中的数据，如果要允许多个node共享一个data目录，需要设置node.max\_local\_storage\_nodes为一个超过1的数字。

## Elasticsearch 性能调优-慢查询日志

从这一讲开始，会去详解es的性能优化这一块，首先如果你要优化性能的话，先要识别出来哪个地方的性能不太如意

es里面的操作，主要分为两种，一种写入（增删改），另一种是查询（搜索）

我们分别要识别出来，哪些写入操作性能比较慢，哪些查询操作性能比较慢，先要识别出来有性能问题的这些慢查询，慢写入，然后才能去考虑如何优化写入的性能，如何优化搜索的性能

在咱们的es学员群里，有时会有人反映，说es性能不行，一次搜索，或者一次聚合，一下子就要10s，或者十几秒，不能接受

搜索慢查询日志

无论是慢查询日志，还是慢写入日志，都是针对shard级别的，因为大家应该知道，无论你是执行增删改，还是执行搜索，都是对某个数据执行写入或者是搜索，其实都是到某个shard上面去执行的

shard上面执行的慢的写入或者是搜索，都会记录在针对这个shard的日志中

阈值的意思，就是说，什么叫做慢？搜索，5s叫做慢？还是10s叫做慢？或者是1s叫做慢？

比如说，你设置一个阈值，5s就是搜索的阈值，5s就叫做慢，那么一旦一个搜索请求超过了5s之后，就会记录一条慢搜索日志到日志文件中

shard level的搜索慢查询日志，辉将搜索性能较慢的查询写入一个专门的日志文件中。可以针对query phase和fetch phase单独设置慢查询的阈值，而具体的慢查询阈值设置如下所示：

在elasticsearch.yml中，设置

index.search.slowlog.threshold.query.warn: 10s

index.search.slowlog.threshold.query.info: 5s

index.search.slowlog.threshold.query.debug: 2s

index.search.slowlog.threshold.query.trace: 500ms

index.search.slowlog.threshold.fetch.warn: 1s

index.search.slowlog.threshold.fetch.info: 800ms

index.search.slowlog.threshold.fetch.debug: 500ms

index.search.slowlog.threshold.fetch.trace: 200ms

而慢查询日志具体的格式，都是在log4j2.properties中配置的，比如下面的配置：

appender.index\_search\_slowlog\_rolling.type = RollingFile

appender.index\_search\_slowlog\_rolling.name = index\_search\_slowlog\_rolling

appender.index\_search\_slowlog\_rolling.fileName = ${sys:es.logs}\_index\_search\_slowlog.log

appender.index\_search\_slowlog\_rolling.layout.type = PatternLayout

appender.index\_search\_slowlog\_rolling.layout.pattern = [%d{ISO8601}][%-5p][%-25c] %.10000m%n

appender.index\_search\_slowlog\_rolling.filePattern = ${sys:es.logs}\_index\_search\_slowlog-%d{yyyy-MM-dd}.log

appender.index\_search\_slowlog\_rolling.policies.type = Policies

appender.index\_search\_slowlog\_rolling.policies.time.type = TimeBasedTriggeringPolicy

appender.index\_search\_slowlog\_rolling.policies.time.interval = 1

appender.index\_search\_slowlog\_rolling.policies.time.modulate = true

logger.index\_search\_slowlog\_rolling.name = index.search.slowlog

logger.index\_search\_slowlog\_rolling.level = trace

logger.index\_search\_slowlog\_rolling.appenderRef.index\_search\_slowlog\_rolling.ref = index\_search\_slowlog\_rolling

logger.index\_search\_slowlog\_rolling.additivity = false

索引慢写入日志

可以用如下的配置来设置索引写入慢日志的阈值：

index.indexing.slowlog.threshold.index.warn: 10s

index.indexing.slowlog.threshold.index.info: 5s

index.indexing.slowlog.threshold.index.debug: 2s

index.indexing.slowlog.threshold.index.trace: 500ms

index.indexing.slowlog.level: info

index.indexing.slowlog.source: 1000

用下面的log4j.properties配置就可以设置索引慢写入日志的格式：

appender.index\_indexing\_slowlog\_rolling.type = RollingFile

appender.index\_indexing\_slowlog\_rolling.name = index\_indexing\_slowlog\_rolling

appender.index\_indexing\_slowlog\_rolling.fileName = ${sys:es.logs}\_index\_indexing\_slowlog.log

appender.index\_indexing\_slowlog\_rolling.layout.type = PatternLayout

appender.index\_indexing\_slowlog\_rolling.layout.pattern = [%d{ISO8601}][%-5p][%-25c] %marker%.10000m%n

appender.index\_indexing\_slowlog\_rolling.filePattern = ${sys:es.logs}\_index\_indexing\_slowlog-%d{yyyy-MM-dd}.log

appender.index\_indexing\_slowlog\_rolling.policies.type = Policies

appender.index\_indexing\_slowlog\_rolling.policies.time.type = TimeBasedTriggeringPolicy

appender.index\_indexing\_slowlog\_rolling.policies.time.interval = 1

appender.index\_indexing\_slowlog\_rolling.policies.time.modulate = true

logger.index\_indexing\_slowlog.name = index.indexing.slowlog.index

logger.index\_indexing\_slowlog.level = trace

logger.index\_indexing\_slowlog.appenderRef.index\_indexing\_slowlog\_rolling.ref = index\_indexing\_slowlog\_rolling

logger.index\_indexing\_slowlog.additivity = false

把这个东西记录好了以后，就可以什么呢？es集群运维管理员，就需要经常每天去看看，正常情况下，慢查询应该是比较少数的。所以比如你每天检查一次，如果发现某个查询特别慢，就要通知写这个查询的RD，开发人员，让他们去优化一下性能。

## Elasticsearch 性能调优-基本优化建议

最最基础，要成为同学脑海中，对es的性能优化的一种常识性的知识，以后做es的时候，就是要按照这个最基本的规范来干

1、搜索结果不要返回过大的结果集

es是一个搜索引擎，所以如果用这个搜索引擎对大量的数据进行搜索，并且返回搜索结果中排在最前面的少数结果，是非常合适的。然而，如果要做成类似数据库的东西，每次都进行大批量的查询，是很不合适的。如果真的要做大批量结果的查询，记得考虑用scroll api。

2、避免超大的document

http.max\_context\_length的默认值是100mb，意味着你一次document写入时，document的内容不能超过100mb，否则es就会拒绝写入。也许你可以将这个参数设置的更大，从而让你的超大的documdent可以写入es，但是es底层的lucene引擎还是有一个2gb的最大限制。

即使我们不考虑引擎层的限制，超大的document在实际生产环境中是很不好的。超大document会耗费更多的网络资源，内存资源和磁盘资源，甚至对那些不要求获取\_source的请求，也是一样，因为es需要从\_source中提取\_id字段，对于超大document这个获取\_id字段的过程的资源开销也是很大的。而将这种超大document写入es也会使用大量的内存，占用内存空间的大小甚至会是documdent本身大小的数倍。近似匹配的搜索，比如phrase query，以及高亮显示，对超大document的资源开销会更大，因为这些操作的性能开销直接跟document的大小成正比。

因此对于超大document，我们需要考虑一下，我们到底需要其中的哪些部分。举例来说，如果我们要对一些书进行搜索，那么我们并不需要将整本书的内容就放入es中吧。我们可以仅仅使用每一篇章或者一个段落作为一个document，然后给一个field标识出来这些document属于哪本书，这样每个document的大小不就变小了么。这就可以避免超大document导致的各种开销，同时可以优化搜索的体验。比如说，如果一个用户要搜索两个单词，foo和bar，如果在两个不同的段落中分别匹配了一个单词，肯定匹配效果要比，一个段落中匹配了两个单词，要差。

3、避免稀疏的数据

lucene的内核结构，跟稠密的数据配合起来，性能会更好，举个例子，什么叫稀疏的数据，什么叫稠密的数据？比如有100个document，每个document都有20个field，20个field都有值，这就是稠密的数据。但是如果100个document，每个document的field都不一样，有的document有2个field，有的document有50个field，这就是稀疏的数据。

原因就是，lucene在内部会通过doc id来唯一标识一个document，这个doc id是integer类型，范围在0到索引中含有的document数量之间。这些doc id是用来在lucene内部的api之间进行通信的，比如说，对一个term用一个match query来进行搜索，就会产生一个doc id集合，然后这些doc id会用来获取对应的norm值，以用来计算每个doc的相关度分数。而根据doc id查找norm的过程，是通过每个document的每个field保留一个字节来进行的一个算法，这个过程叫做norm查找，norm就是每个document的每个field保留的一个字节。对于每个doc id对应的那个norm值，可以通过读取es一个内置索引，叫做doc\_id的索引，中的一个字节来获取。这个过程是性能很高的，而且可以帮助lucene快速的定位到每个document的norm值，但是同时这样的话document本身就不需要存储这一个字节的norm值了。

在实际运行过程中，这就意味着，如果一个索引有100个document，对于每个field，就需要100个字节来存储norm值，即使100个document中只有10个document含有某个field，但是对那个field来说，还是要100个字节来存储norm值。这就会对存储产生更大的开销，存储空间被浪费的一个问题，而且也会影响读写性能。

下面有一些避免稀疏数据的办法：

（1）避免将没有任何关联性的数据写入同一个索引

我们必须避免将结构完全不一样的数据写入同一个索引中，因为结构完全不一样的数据，field是完全不一样的，会导致index数据非常稀疏。最好将这种数据写入不同的索引中，如果这种索引数据量比较少，那么可以考虑给其很少的primary shard，比如1个，避免资源浪费。

（2）对document的结构进行规范化/标准化

即使我们真的要将不同类型的document写入相同的索引中，还是有办法可以避免稀疏性，那就是对不同类型的document进行标准化。比如说，如果所有的document都有一个时间戳field，不过有的叫做timestamp，有的叫做creation\_date，那么可以将不同document的这个field重命名为相同的字段，尽量让documment的结构相同。另外一个，就是比如有的document有一个字段，叫做goods\_type，但是有的document没有这个字段，此时可以对没有这个字段的document，补充一个goods\_type给一个默认值，比如default。

（3）避免使用多个types存储不一样结构的document

很多人会很喜欢在一个index中放很多个types来存储不同类型的数据。但是其实不是这样的，最好不要这么干，如果你在一个index中有多个type，但是这些type的数据结构不太一样，那么这些type实际上底层都是写到这个索引中的，还是会导致稀疏性。如果多个type的结构不太一样，最好放入不同的索引中，不要写入一个索引中。

（4）对稀疏的field禁用norms和doc\_values

如果上面的步骤都没法做，那么只能对那种稀疏的field，禁止norms和doc\_values字段，因为这两个字段的存储机制类似，都是每个field有一个全量的存储，对存储浪费很大。如果一个field不需要考虑其相关度分数，那么可以禁用norms，如果不需要对一个field进行排序或者聚合，那么可以禁用doc\_values字段。

## Elasticsearch 性能调优-索引写入性能优化

1和2，适合的是，你的es java client程序，可以采取批量写的场景

3，比较通用的，比较合适的是，你对于写入数据到可以读到能够接受比较大的延迟

4，一次性批量导入数据的场景

5/6/7/8/9，通用型，尽量都去做到

1/2/3/4，都是有各自适用的场景，如果场景合适，就尽量用，因为对性能的提升都是很明显的

5/6/7/8/9，其中通用，尽量去优化你的集群，但是其中最重要的，就是3块，filesystem cache更大的内存，给index buffer最充足的内存，一个是用SSD固态硬盘

1、用bulk批量写入

你如果要往es里面灌入数据的话，那么根据你的业务场景来，如果你的业务场景可以支持，可以做到，让你将一批数据聚合起来，一次性写入es，那么就尽量采用bulk的方式，每次批量写个几百条这样子。

bulk批量写入的性能比你一条一条写入大量的document的性能要好很多。但是如果要知道一个bulk请求最佳的大小，需要对单个es node的单个shard做压测。先bulk写入100个document，然后200个，400个，以此类推，每次都将bulk size加倍一次。如果bulk写入性能开始变平缓的时候，那么这个就是最佳的bulk大小。并不是bulk size越大越好，而是根据你的集群等环境具体要测试出来的，因为越大的bulk size会导致内存压力过大，因此最好一个请求不要发送超过10mb的数据量。

之前有es学员就是在公司里测试这个bulk写入，上来就是多线程并发写bulk，但是这里面就有一个问题，刚开始，你先确定一个是bulk size，此时就尽量是用你的程序，单线程，一个es node，一个shard，测试。看看单线程最多一次性写多少条数据，性能是比较好的。

2、使用多线程将数据写入es

单线程发送bulk请求是无法最大化es集群写入的吞吐量的。如果要利用集群的所有资源，就需要使用多线程并发将数据bulk写入集群中。为了更好的利用集群的资源，这样多线程并发写入，可以减少每次底层磁盘fsync的次数和开销。一样，可以对单个es节点的单个shard做压测，比如说，先是2个线程，然后是4个线程，然后是8个线程，16个，每次线程数量倍增。一旦发现es返回了TOO\_MANY\_REQUESTS的错误，JavaClient也就是EsRejectedExecutionException，之前有学员就是做多线程的bulk写入的时候，就发生了。此时那么就说明es是说已经到了一个并发写入的最大瓶颈了，此时我们就知道最多只能支撑这么高的并发写入了。

3、增加refresh间隔

默认的refresh间隔是1s，用index.refresh\_interval参数可以设置，这样会其强迫es每秒中都将内存中的数据写入磁盘中，创建一个新的segment file。正是这个间隔，让我们每次写入数据后，1s以后才能看到。但是如果我们将这个间隔调大，比如30s，可以接受写入的数据30s后才看到，那么我们就可以获取更大的写入吞吐量，因为30s内都是写内存的，每隔30s才会创建一个segment file。

4、禁止refresh和replia

如果我们要一次性加载大批量的数据进es，可以先禁止refresh和replia复制，将index.refresh\_interval设置为-1，将index.number\_of\_replicas设置为0即可。这可能会导致我们的数据丢失，因为没有refresh和replica机制了。但是不需要创建segment file，也不需要将数据replica复制到其他的replica shasrd上面去。此时写入的速度会非常快，一旦写完之后，可以将refresh和replica修改回正常的状态。

5、禁止swapping交换内存

之前讲解果，可以将swapping禁止掉，有的时候，如果要将es jvm内存交换到磁盘，再交换回内存，大量磁盘IO，性能很差

6、给filesystem cache更多的内存

filesystem cache被用来执行更多的IO操作，如果我们能给filesystem cache更多的内存资源，那么es的写入性能会好很多。

7、使用自动生成的id

如果我们要手动给es document设置一个id，那么es需要每次都去确认一下那个id是否存在，这个过程是比较耗费时间的。如果我们使用自动生成的id，那么es就可以跳过这个步骤，写入性能会更好。对于你的业务中的表id，可以作为es document的一个field。

8、用性能更好的硬件

我们可以给filesystem cache更多的内存，也可以使用SSD替代机械硬盘，避免使用NAS等网络存储，考虑使用RAID 0来条带化存储提升磁盘并行读写效率，等等。

9、index buffer

如果我们要进行非常重的高并发写入操作，那么最好将index buffer调大一些，indices.memory.index\_buffer\_size，这个可以调节大一些，设置的这个index buffer大小，是所有的shard公用的，但是如果除以shard数量以后，算出来平均每个shard可以使用的内存大小，一般建议，但是对于每个shard来说，最多给512mb，因为再大性能就没什么提升了。es会将这个设置作为每个shard共享的index buffer，那些特别活跃的shard会更多的使用这个buffer。默认这个参数的值是10%，也就是jvm heap的10%，如果我们给jvm heap分配10gb内存，那么这个index buffer就有1gb，对于两个shard共享来说，是足够的了。

## Elasticsearch 性能调优-搜索性能优化

magic，如果真的要优化搜索性能的话，就是以下几种办法

1/5，配合起来，就是搜索性能优化的杀手锏

3/4，配合起来，解决各种复杂的搜索需求的性能

1、给filesysgtem cache更多的内存

es的搜索引擎严重依赖于底层的filesystem cache，你如果给filesystem cache更多的内存，尽量让内存可以容纳所有的indx segment file索引数据文件，那么你搜索的时候就基本都是走内存的，性能会非常高。

之前有个学员，一直在问我，说他的搜索性能，聚合性能，倒排索引，正排索引，磁盘文件，十几秒。。。。

比如说，你，es节点有3台机器，每台机器，看起来内存很多，64G，总内存，64 \* 3

每台机器给es jvm heap是32G，那么剩下来留给filesystem cache的就是每台机器才32g，总共集群里给filesystem cache的就是32 \* 3 = 96gb内存

如果你此时，你整个，磁盘上索引数据文件，在3台机器上，一共占用了1T的磁盘容量，你的es数据量是1t

你觉得你的性能能好吗？filesystem cache的内存才100g，十分之一的数据可以放内存，其他的都在磁盘，然后你执行搜索操作，大部分操作都是走磁盘，性能肯定差

归根结底，你要让es性能要好，最佳的情况下，就是你的机器的内存，至少可以容纳你的总数据量的一半

比如说，你一共要在es中存储1T的数据，那么你的多台机器留个filesystem cache的内存加起来综合，至少要到512G，至少半数的情况下，搜索是走内存的，性能一般可以到几秒钟，2秒，3秒，5秒

如果最佳的情况下，我们自己的生产环境实践经验，最好是用es就存少量的数据，就是你要用来搜索的那些索引，内存留给filesystem cache的，就100G，那么你就控制在100gb以内，相当于是，你的数据几乎全部走内存来搜索，性能非常之高，一般可以在1秒以内

所以之前有些学员也是问，我也是跟他们说，尽量在es里，就存储必须用来搜索的数据，比如说你现在有一份数据，有100个字段，其实用来搜索的只有10个字段，建议是将10个字段的数据，存入es，剩下90个字段的数据，可以放mysql，hadoop hbase，都可以

这样的话，es数据量很少，10个字段的数据，都可以放内存，就用来搜索，搜索出来一些id，通过id去mysql，hbase里面去查询明细的数据

2、用更快的硬件资源

（1）给filesystem cache更多的内存资源

（2）用SSD固态硬盘

（3）使用本地存储系统，不要用NFS等网络存储系统

（4）给更多的CPU资源

3、document模型设计

document模型设计是非常重要的，很多操作，不要在搜索的时候才想去执行各种复杂的乱七八糟的操作。es能支持的操作就是那么多，不要考虑用es做一些它不好操作的事情。如果真的有那种操作，尽量在document模型设计的时候，写入的时候就完成。另外对于一些太复杂的操作，比如join，nested，parent-child搜索都要尽量避免，性能都很差的。

很多同学在问我，很多复杂的乱七八糟的一些操作，如何执行

两个思路，在搜索/查询的时候，要执行一些业务强相关的特别复杂的操作：

（1）在写入数据的时候，就设计好模型，加几个字段，把处理好的数据写入加的字段里面

（2）自己用java程序封装，es能做的，用es来做，搜索出来的数据，在java程序里面去做，比如说我们，基于es，用java封装一些特别复杂的操作

4、预先index data

为了性能，提前优化data index时的数据模型，比如说document有一个price field，然后大多数查询都对一个固定的范围，对这个field使用range范围查询，那么可以提前将这个price的范围处理出来，写入一个字段中。比如下面这样：

PUT index/type/1

{

"designation": "spoon",

"price": 13

}

GET index/\_search

{

"aggs": {

"price\_ranges": {

"range": {

"field": "price",

"ranges": [

{ "to": 10 },

{ "from": 10, "to": 100 },

{ "from": 100 }

]

}

}

}

}

我们完全可以增加一个price\_range字段：

PUT index

{

"mappings": {

"type": {

"properties": {

"price\_range": {

"type": "keyword"

}

}

}

}

}

然后写入的时候，直接计算出来这个range：

PUT index/type/1

{

"designation": "spoon",

"price": 13,

"price\_range": "10-100"

}

然后搜索的时候，就可以直接用term查询了，性能非常高：

GET index/\_search

{

"aggs": {

"price\_ranges": {

"terms": {

"field": "price\_range"

}

}

}

}

5、预热filesystem cache

如果我们重启了es，那么filesystem cache是空壳的，就需要不断的查询才能重新让filesystem cache热起来，我们可以先说动对一些数据进行查询。

比如说，你本来一个查询，要用户点击以后才执行，才能从磁盘加载到filesystem cache里，第一次执行要10s，以后每次就几百毫秒

你完全可以，自己早上的时候，就程序执行那个查询，预热，数据就加载到filesystem cahce，程序执行的时候是10s，以后用户真的来看的时候就才几百毫秒

6、避免使用script脚本

说实话，一般是避免使用es script的，实际生产中更是少用，性能不高，尽量不要使用

7、使用固定范围的日期查询

尽量不要使用now这种内置函数来执行日期查询，因为默认now是到毫秒级的，是无法缓存结果，尽量使用一个阶段范围，比如now/m，就是到分钟级

那么如果一分钟内，都执行这个查询，是可以取用查询缓存的

PUT index/type/1

{

"my\_date": "2016-05-11T16:30:55.328Z"

}

GET index/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"my\_date": {

"gte": "now-1h",

"lte": "now"

}

}

}

}

}

}

完全可以替换为：

GET index/\_search

{

"query": {

"constant\_score": {

"filter": {

"range": {

"my\_date": {

"gte": "now-1h/m",

"lte": "now/m"

}

}

}

}

}

}

## Elasticsearch 性能调优-磁盘读写性能优化

优化磁盘空间的占用，减少磁盘空间的占用，更多的数据可以进入filesystem cache

比如说你原来，磁盘空间占用一共是1T，内存只有512G，现在优化了磁盘空间占用之后，减少了数据量，可能数据量就只有512G了，那么就可以全部进入内存

1、禁用不需要的功能

聚合，搜索，评分，近似匹配

聚合：doc values

搜索：倒排索引，index

评分：norms

近似匹配：index\_options（freqs）

任何一个功能不需要，就把对应的存储的数据给干掉，这样可以节约磁盘空间的占用，也可以优化磁盘的读写性能

默认情况下，es在写入document到索引的时候，都会给大多数的field增加一份doc values，就是正排索引，用来进行聚合或者排序的。比如说，如果我们有一个叫做foo的数字类型field，我们要对这个字段运行histograms aggr聚合操作，但是可能我们并不需要对这个字段进行搜索，那么就可以禁止为这个字段生成倒排索引，只需要doc value正排索引即可。禁用倒排索引：

PUT index

{

"mappings": {

"type": {

"properties": {

"foo": {

"type": "integer",

"index": false

}

}

}

}

}

text类型的field会存储norm值，用来计算doc的相关度分数，如果我们需要对一个text field进行搜索，但是不关心这个field的分数，那么可以禁用norm值

PUT index

{

"mappings": {

"type": {

"properties": {

"foo": {

"type": "text",

"norms": false

}

}

}

}

}

text field还会存储出现频率以及位置，出现频率也是用来计算相关度分数的，位置是用来进行phrase query这种近似匹配操作的，如果我们不需要执行phrase query近似匹配，那么可以禁用位置这个属性：

PUT index

{

"mappings": {

"type": {

"properties": {

"foo": {

"type": "text",

"index\_options": "freqs"

}

}

}

}

}

此外，如果我们不关心相关度频分，我们可以配置es仅仅为每个term索引对应的document，我们可以对这个field进行搜索，但是phrase query这种近似匹配会报错，而且相关度评分会不准确：

PUT index

{

"mappings": {

"type": {

"properties": {

"foo": {

"type": "text",

"norms": false,

"index\_options": "freqs"

}

}

}

}

}

2、不要用默认的动态string类型映射

默认的动态string类型映射会将string类型的field同时映射为text类型以及keyword类型，这会浪费磁盘空间，因为我们不一定两种都需要。通常来说，id field这种字段可能只需要keyword映射，而body field可能只需要text field。

映射一个content，content: text，content.内置字段: keyword

可以通过手动设置mappings映射来避免字符串类型的field被自动映射为text和keyword：

PUT index

{

"mappings": {

"type": {

"dynamic\_templates": [

{

"strings": {

"match\_mapping\_type": "string",

"mapping": {

"type": "keyword"

}

}

}

]

}

}

}

3、禁止\_all field

\_all field会将document中所有field的值都合并在一起进行索引，很耗费空空间，如果不需要一次性对所有的field都进行搜索，那么最好禁用\_all field。

4、使用best\_compression

\_source field和其他field都很耗费磁盘空间，最好是对其使用best\_compression进行压缩。用elasticsearch.yml中的index.codec来设置，将其设置为best\_compression即可。

5、用最小的最合适的数字类型

es支持4种数字类型，byte，short，integer，long。如果最小的类型就合适，那么就用最小的类型。

## ES 生产集群监控-基于 cat api 进行监控

es集群监控，最好是自己干吧，因为官方出了那种非常棒的x-pack做权限认证，监控，等等，做的都非常好，但是。。。是收费的。。。

自己做es集群监控，就是根据es的一些api，自己写一个java web的应用，自己做前端界面，程序里不断的每隔几秒钟，调用一次后端的接口，获取到各种监控信息，然后用前端页面显示出来，开发开发一个可视化的es集群的监控的工作台

在es老版本，有一个很好用的插件，叫做head，但是5.x之后都收口了，不让做这种插件了，主推自己的x-pack，要收费，要盈利，赚钱

1、GET /\_cat/aliases?v

看到集群中有哪些索引别名

alias index filter routing.index routing.search

alias1 test1 - - -

alias2 test1 \* - -

alias3 test1 - 1 1

alias4 test1 - 2 1,2

2、GET /\_cat/allocation?v

看到每个节点分配了几个shard，对磁盘的占用空间大小，使用率，等等

shards disk.indices disk.used disk.avail disk.total disk.percent host ip node

5 260b 47.3gb 43.4gb 100.7gb 46 127.0.0.1 127.0.0.1 CSUXak2

3、GET /\_cat/count?v

看每个索引的document数量

epoch timestamp count

1475868259 15:24:20 120

4、GET /\_cat/fielddata?v

看每个节点的jvm heap内存中的fielddata内存占用情况（对分词的field进行聚合/排序要用jvm heap中的正排索引，fielddata）

id host ip node field size

Nqk-6inXQq-OxUfOUI8jNQ 127.0.0.1 127.0.0.1 Nqk-6in body 544b

Nqk-6inXQq-OxUfOUI8jNQ 127.0.0.1 127.0.0.1 Nqk-6in soul 480b

5、GET /\_cat/health?v

比较全面的看一个es集群的整体健康状况，主要是看是green，yellow，red

epoch timestamp cluster status node.total node.data shards pri relo init unassign pending\_tasks max\_task\_wait\_time active\_shards\_percent

1475871424 16:17:04 elasticsearch green 1 1 5 5 0 0 0 0 - 100.0%

6、GET /\_cat/indices?v

每个索引的具体的情况，比如有几个shard，多少个document，被删除的document有多少，占用了多少磁盘空间

health status index uuid pri rep docs.count docs.deleted store.size pri.store.size

yellow open twitter u8FNjxh8Rfy\_awN11oDKYQ 1 1 1200 0 88.1kb 88.1kb

7、GET /\_cat/master?v

看master node当前的具体的情况，哪个node是当前的master node

id host ip node

YzWoH\_2BT-6UjVGDyPdqYg 127.0.0.1 127.0.0.1 YzWoH\_2

8、GET /\_cat/nodes?v

看每个node的具体的情况，就比如jvm heap内存使用率，内存使用率，cpu load，是什么角色

ip heap.percent ram.percent cpu load\_1m load\_5m load\_15m node.role master name

127.0.0.1 65 99 42 3.07 mdi \* mJw06l1

9、GET /\_cat/pending\_tasks?v

看当前pending没执行完的task的具体情况，执行的是什么操作

insertOrder timeInQueue priority source

1685 855ms HIGH update-mapping [foo][t]

1686 843ms HIGH update-mapping [foo][t]

1693 753ms HIGH refresh-mapping [foo][[t]]

1688 816ms HIGH update-mapping [foo][t]

1689 802ms HIGH update-mapping [foo][t]

1690 787ms HIGH update-mapping [foo][t]

1691 773ms HIGH update-mapping [foo][t]

10、GET /\_cat/plugins?v&s=component&h=name,component,version,description

看当前集群安装了哪些插件

name component version description

U7321H6 analysis-icu 5.5.1 The ICU Analysis plugin integrates Lucene ICU module into elasticsearch, adding ICU relates analysis components.

U7321H6 analysis-kuromoji 5.5.1 The Japanese (kuromoji) Analysis plugin integrates Lucene kuromoji analysis module into elasticsearch.

U7321H6 analysis-phonetic 5.5.1 The Phonetic Analysis plugin integrates phonetic token filter analysis with elasticsearch.

U7321H6 analysis-smartcn 5.5.1 Smart Chinese Analysis plugin integrates Lucene Smart Chinese analysis module into elasticsearch.

U7321H6 analysis-stempel 5.5.1 The Stempel (Polish) Analysis plugin integrates Lucene stempel (polish) analysis module into elasticsearch.

U7321H6 analysis-ukrainian 5.5.1 The Ukrainian Analysis plugin integrates the Lucene UkrainianMorfologikAnalyzer into elasticsearch.

U7321H6 discovery-azure-classic 5.5.1 The Azure Classic Discovery plugin allows to use Azure Classic API for the unicast discovery mechanism

U7321H6 discovery-ec2 5.5.1 The EC2 discovery plugin allows to use AWS API for the unicast discovery mechanism.

U7321H6 discovery-file 5.5.1 Discovery file plugin enables unicast discovery from hosts stored in a file.

U7321H6 discovery-gce 5.5.1 The Google Compute Engine (GCE) Discovery plugin allows to use GCE API for the unicast discovery mechanism.

U7321H6 ingest-attachment 5.5.1 Ingest processor that uses Apache Tika to extract contents

U7321H6 ingest-geoip 5.5.1 Ingest processor that uses looksup geo data based on ip adresses using the Maxmind geo database

U7321H6 ingest-user-agent 5.5.1 Ingest processor that extracts information from a user agent

U7321H6 jvm-example 5.5.1 Demonstrates all the pluggable Java entry points in Elasticsearch

U7321H6 lang-javascript 5.5.1 The JavaScript language plugin allows to have javascript as the language of scripts to execute.

U7321H6 lang-python 5.5.1 The Python language plugin allows to have python as the language of scripts to execute.

U7321H6 mapper-attachments 5.5.1 The mapper attachments plugin adds the attachment type to Elasticsearch using Apache Tika.

U7321H6 mapper-murmur3 5.5.1 The Mapper Murmur3 plugin allows to compute hashes of a field's values at index-time and to store them in the index.

U7321H6 mapper-size 5.5.1 The Mapper Size plugin allows document to record their uncompressed size at index time.

U7321H6 store-smb 5.5.1 The Store SMB plugin adds support for SMB stores.

11、GET \_cat/recovery?v

看shard recovery恢复的一个过程的具体情况

index shard time type stage source\_host source\_node target\_host target\_node repository snapshot files files\_recovered files\_percent files\_total bytes bytes\_recovered bytes\_percent bytes\_total translog\_ops translog\_ops\_recovered translog\_ops\_percent

twitter 0 13ms store done n/a n/a node0 node-0 n/a n/a 0 0 100% 13 0 0 100% 9928 0 0 100.0%

12、GET /\_cat/repositories?v

查看用于snapshotting的repository有哪些

id type

repo1 fs

repo2 s3

13、GET /\_cat/thread\_pool?v

看每个线程池的具体的情况

Z6MkIvC bulk 0 0 0

Z6MkIvC fetch\_shard\_started 0 0 0

Z6MkIvC fetch\_shard\_store 0 0 0

Z6MkIvC flush 0 0 0

Z6MkIvC force\_merge 0 0 0

Z6MkIvC generic 0 0 0

Z6MkIvC get 0 0 0

Z6MkIvC index 0 0 0

Z6MkIvC listener 0 0 0

Z6MkIvC management 1 0 0

Z6MkIvC refresh 0 0 0

Z6MkIvC search 0 0 0

Z6MkIvC snapshot 0 0 0

Z6MkIvC warmer 0 0 0

14、GET \_cat/shards?v

看每个shard的具体的情况

twitter 0 p STARTED 3014 31.1mb 192.168.56.10 H5dfFeA

twitter 0 r UNASSIGNED

15、GET /\_cat/segments?v

看每个segement，索引segment文件的情况，在哪个node上，有多少个document，占用了多少磁盘空间，有多少数据在内存中，是否可以搜索

index shard prirep ip segment generation docs.count docs.deleted size size.memory committed searchable version compound

test 3 p 127.0.0.1 \_0 0 1 0 3kb 2042 false true 6.5.1 true

test1 3 p 127.0.0.1 \_0 0 1 0 3kb 2042 false true 6.5.1 true

16、GET /\_cat/snapshots?v&s=id

看当前执行的snapshot的操作

id status start\_epoch start\_time end\_epoch end\_time duration indices successful\_shards failed\_shards total\_shards

snap1 FAILED 1445616705 18:11:45 1445616978 18:16:18 4.6m 1 4 1 5

snap2 SUCCESS 1445634298 23:04:58 1445634672 23:11:12 6.2m 2 10 0 10

17、GET /\_cat/templates?v&s=name

看当前有的那些tempalte，具体的情况是什么

name template order version

template0 te\* 0

template1 tea\* 1

template2 teak\* 2 7

## ES 生产集群监控-基于 cluster api 进行监控

1、GET \_cluster/health

{

"cluster\_name" : "testcluster",

"status" : "yellow",

"timed\_out" : false,

"number\_of\_nodes" : 1,

"number\_of\_data\_nodes" : 1,

"active\_primary\_shards" : 5,

"active\_shards" : 5,

"relocating\_shards" : 0,

"initializing\_shards" : 0,

"unassigned\_shards" : 5,

"delayed\_unassigned\_shards": 0,

"number\_of\_pending\_tasks" : 0,

"number\_of\_in\_flight\_fetch": 0,

"task\_max\_waiting\_in\_queue\_millis": 0,

"active\_shards\_percent\_as\_number": 50.0

}

2、GET \_cluster/stats?human&pretty

{

"timestamp": 1459427693515,

"cluster\_name": "elasticsearch",

"status": "green",

"indices": {

"count": 2,

"shards": {

"total": 10,

"primaries": 10,

"replication": 0,

"index": {

"shards": {

"min": 5,

"max": 5,

"avg": 5

},

"primaries": {

"min": 5,

"max": 5,

"avg": 5

},

"replication": {

"min": 0,

"max": 0,

"avg": 0

}

}

},

"docs": {

"count": 10,

"deleted": 0

},

"store": {

"size": "16.2kb",

"size\_in\_bytes": 16684,

"throttle\_time": "0s",

"throttle\_time\_in\_millis": 0

},

"fielddata": {

"memory\_size": "0b",

"memory\_size\_in\_bytes": 0,

"evictions": 0

},

"query\_cache": {

"memory\_size": "0b",

"memory\_size\_in\_bytes": 0,

"total\_count": 0,

"hit\_count": 0,

"miss\_count": 0,

"cache\_size": 0,

"cache\_count": 0,

"evictions": 0

},

"completion": {

"size": "0b",

"size\_in\_bytes": 0

},

"segments": {

"count": 4,

"memory": "8.6kb",

"memory\_in\_bytes": 8898,

"terms\_memory": "6.3kb",

"terms\_memory\_in\_bytes": 6522,

"stored\_fields\_memory": "1.2kb",

"stored\_fields\_memory\_in\_bytes": 1248,

"term\_vectors\_memory": "0b",

"term\_vectors\_memory\_in\_bytes": 0,

"norms\_memory": "384b",

"norms\_memory\_in\_bytes": 384,

"doc\_values\_memory": "744b",

"doc\_values\_memory\_in\_bytes": 744,

"index\_writer\_memory": "0b",

"index\_writer\_memory\_in\_bytes": 0,

"version\_map\_memory": "0b",

"version\_map\_memory\_in\_bytes": 0,

"fixed\_bit\_set": "0b",

"fixed\_bit\_set\_memory\_in\_bytes": 0,

"file\_sizes": {}

},

"percolator": {

"num\_queries": 0

}

},

"nodes": {

"count": {

"total": 1,

"data": 1,

"coordinating\_only": 0,

"master": 1,

"ingest": 1

},

"versions": [

"5.5.1"

],

"os": {

"available\_processors": 8,

"allocated\_processors": 8,

"names": [

{

"name": "Mac OS X",

"count": 1

}

],

"mem" : {

"total" : "16gb",

"total\_in\_bytes" : 17179869184,

"free" : "78.1mb",

"free\_in\_bytes" : 81960960,

"used" : "15.9gb",

"used\_in\_bytes" : 17097908224,

"free\_percent" : 0,

"used\_percent" : 100

}

},

"process": {

"cpu": {

"percent": 9

},

"open\_file\_descriptors": {

"min": 268,

"max": 268,

"avg": 268

}

},

"jvm": {

"max\_uptime": "13.7s",

"max\_uptime\_in\_millis": 13737,

"versions": [

{

"version": "1.8.0\_74",

"vm\_name": "Java HotSpot(TM) 64-Bit Server VM",

"vm\_version": "25.74-b02",

"vm\_vendor": "Oracle Corporation",

"count": 1

}

],

"mem": {

"heap\_used": "57.5mb",

"heap\_used\_in\_bytes": 60312664,

"heap\_max": "989.8mb",

"heap\_max\_in\_bytes": 1037959168

},

"threads": 90

},

"fs": {

"total": "200.6gb",

"total\_in\_bytes": 215429193728,

"free": "32.6gb",

"free\_in\_bytes": 35064553472,

"available": "32.4gb",

"available\_in\_bytes": 34802409472

},

"plugins": [

{

"name": "analysis-icu",

"version": "5.5.1",

"description": "The ICU Analysis plugin integrates Lucene ICU module into elasticsearch, adding ICU relates analysis components.",

"classname": "org.elasticsearch.plugin.analysis.icu.AnalysisICUPlugin",

"has\_native\_controller": false

},

{

"name": "ingest-geoip",

"version": "5.5.1",

"description": "Ingest processor that uses looksup geo data based on ip adresses using the Maxmind geo database",

"classname": "org.elasticsearch.ingest.geoip.IngestGeoIpPlugin",

"has\_native\_controller": false

},

{

"name": "ingest-user-agent",

"version": "5.5.1",

"description": "Ingest processor that extracts information from a user agent",

"classname": "org.elasticsearch.ingest.useragent.IngestUserAgentPlugin",

"has\_native\_controller": false

}

]

}

}

3、GET \_cluster/pending\_tasks

{

"tasks": [

{

"insert\_order": 101,

"priority": "URGENT",

"source": "create-index [foo\_9], cause [api]",

"time\_in\_queue\_millis": 86,

"time\_in\_queue": "86ms"

},

{

"insert\_order": 46,

"priority": "HIGH",

"source": "shard-started ([foo\_2][1], node[tMTocMvQQgGCkj7QDHl3OA], [P], s[INITIALIZING]), reason [after recovery from shard\_store]",

"time\_in\_queue\_millis": 842,

"time\_in\_queue": "842ms"

},

{

"insert\_order": 45,

"priority": "HIGH",

"source": "shard-started ([foo\_2][0], node[tMTocMvQQgGCkj7QDHl3OA], [P], s[INITIALIZING]), reason [after recovery from shard\_store]",

"time\_in\_queue\_millis": 858,

"time\_in\_queue": "858ms"

}

]

}

## 补充案例-环境准备：虚拟机环境搭建

我们从这一讲开始更新两个小型的项目案例，作为咱们这个es课程的终结

入门，高级，运维，项目，带一带，用es可以做什么样的项目，2个小型的案例，走一遍

CentOS 7虚拟机

1、在虚拟机中安装CentOS

启动一个virtual box虚拟机管理软件（vmware，我早些年，发现不太稳定，主要是当时搭建一个hadoop大数据的集群，发现每次休眠以后再重启，集群就挂掉了）

virtual box，发现很稳定，集群从来不会随便乱挂，所以就一直用virtual box了

安装virtual box

用的是什么centos镜像，CentOS比较新的版本是7了，然后服务器上装操作系统的话，内存一般比较大，一般是安装64位的，32位的有一个最大内存4G的限制

（1）使用课程提供的CentOS 7镜像即可，CentOS-7-x86\_64-Minimal-1611.iso。

（2）创建虚拟机：打开Virtual Box，点击“新建”按钮，点击“下一步”，输入虚拟机名称为elasticsearch01，选择操作系统为Linux，选择版本为Red Hat-64bit，分配4096MB内存，后面的选项全部用默认，在Virtual Disk File location and size中，一定要自己选择一个目录来存放虚拟机文件，最后点击“create”按钮，开始创建虚拟机。

（3）设置虚拟机网卡：选择创建好的虚拟机，点击“设置”按钮，在网络一栏中，连接方式中，选择“Bridged Adapter”。

（4）安装虚拟机中的CentOS 7操作系统：选择创建好的虚拟机，点击“开始”按钮，选择安装介质（即本地的CentOS 7镜像文件），按照课程选择后自动安装即可

（5）安装完以后，CentOS会提醒你要重启一下，就是reboot，你就reboot就可以了。

（6）配置网络

vi /etc/sysconfig/network-scripts/ifcfg-enp0s3

先让它动态分配一个ip地址

ONBOOT=yes

service network restart

ip addr

再设置静态ip地址

BOOTPROTO=static

IPADDR=192.168.31.250

NETMASK=255.255.255.0

GATEWAY=192.168.31.1

service network restart

ip addr

配置DNS

检查NetManager的状态：systemctl status NetworkManager.service

检查NetManager管理的网络接口：nmcli dev status

检查NetManager管理的网络连接：nmcli connection show

设置dns：nmcli con mod enp0s3 ipv4.dns "114.114.114.114 8.8.8.8"

让dns配置生效：nmcli con up enp0s3

（7）配置hosts

vi /etc/hosts

配置本机的hostname到ip地址的映射

（8）配置SecureCRT

此时就可以使用SecureCRT从本机连接到虚拟机进行操作了

一般来说，虚拟机管理软件，virtual box，可以用来创建和管理虚拟机，但是一般不会直接在virtualbox里面去操作，因为比较麻烦，没有办法复制粘贴

SecureCRT，在windows宿主机中，去连接virtual box中的虚拟机

收费的，我这里有完美破解版，跟着课程一起给大家，破解

（9）关闭防火墙

systemctl stop firewalld.service

systemctl disable firewalld.service

关闭windows的防火墙

后面要搭建集群，有的大数据技术的集群之间，在本地你给了防火墙的话，可能会没有办法互相连接，会导致搭建失败

（10）配置yum

yum clean all

yum makecache

yum install wget

------------------------------------------------------------------------------------------

2、在每个CentOS中都安装Java

WinSCP，就是在windows宿主机和linux虚拟机之间互相传递文件的一个工具

（1）安装JDK

1、将jdk-8u131-linux-x64.rpm通过WinSCP上传到虚拟机中

2、安装JDK：rpm -ivh jdk-8u131-linux-x64.rpm

3、配置jdk相关的环境变量

vi .bashrc

export JAVA\_HOME=/usr/java/latest

export PATH=$PATH:$JAVA\_HOME/bin

source .bashrc

4、测试jdk安装是否成功：java -version

## 补充案例-小型流量分析系统：logstash 部署以及上手使用

第一个项目案例，简单介绍一下，es在国内主要用来做什么呢

其中一个es比较重要的一个应用的领域，就是elk做一些数据分析，比如说将一些网站的用户访问的日志，通过logstash，从apache或者nginx服务器上获取到

然后将那些日志文件通过logstash导入到es中

然后通过kibana对es中的数据，进行分析，生成一些报表

小型流量分析系统

elk课程，所以我这里就不重复讲解elk了，直接默认大家对elk都有一定的认识了，如果需要学习elk，可以去龙果看另外一个讲师的课程

如果我这里重复讲解的话，那么没意思，而且对其他课程并不是太好

logstash，简单来说，就是用来从各种各样的数据源，采集数据，mysql，apache日志，mq，将数据进行一些简单的处理，然后将数据写入到其他的一些地方，比如es，mq

1、下载和解压缩logstash

2、logstash pipeline

两个组成部分：input和output，也可以包含一个可选的filter

input plugin负责从数据源中获取数据

filter plugin负责对数据进行定制化的处理和修改

output plugin负责将数据写入目的地中

3、运行最基础的logstash pipeline

bin/logstash -e 'input { stdin { } } output { stdout {} }'

-e：直接在命令行对logstash进行配置，从命令行接受输入，将输出写入命令行

输入：hello world，可以看到输出，logstash会补充timestamp和ip地址

用ctrl-d可以结束这个piepline

## 补充案例-小型流量分析系统：整理流程讲解

小型的流量分析系统，怎么来做，就讲解其中的一个做法和场景

1、一般来说，可以通过hive对昨日的流量日志数据，进行离线批处理，先预先按照维度将一些指标预先聚合出来，将结果写入mysql

做一点特殊的说明，因为这是一个单课，没法深入在hadoop，hive这块展开，所以这块是不讲解的，大家有兴趣的可以自己去看看一些资料

默认有一些预先处理好的数据已经存在于mysql中了

2、我们手动准备一些样例数据，然后写入mysql中，装一个mysql，模拟成是hive导入mysql的一份数据

3、通过logstash，将mysql中的数据导入es中

4、通过kibana+各种es聚合语法，生成各种各样的报表出来

## 补充案例-小型流量分析系统：安装 mysql 以及手动导入样例数据

10 \* 60 \* 1000

用最简单的方式装一个mysql数据库，然后后面的话，就有数据库可以用来开发了

wget http://dev.mysql.com/get/mysql-community-release-el7-5.noarch.rpm

rpm -ivh mysql-community-release-el7-5.noarch.rpm

yum install -y mysql-community-server

service mysqld restart

mysql -u root

set password for 'root'@'localhost' =password('password');

datekey cookie section userid province city pv is\_return\_visit is\_bounce\_visit visit\_time visit\_page\_cnt

日期 cookie 版块 用户id 省份 城市 pv 是否老用户回访 是否跳出 访问时间 访问页面数量

create table user\_access\_log\_aggr (

datekey varchar(255),

cookie varchar(255),

section varchar(255),

userid int,

province varchar(255),

city varchar(255),

pv int,

is\_return\_visit int,

is\_bounce\_visit int,

visit\_time int,

visit\_page\_cnt int

)

insert into user\_access\_log\_aggr values('20171001', 'dasjfkaksdfj33', 'game', 1, 'beijing', 'beijing', 10, 0, 1, 600000, 3);

insert into user\_access\_log\_aggr values('20171001', 'dasjadfssdfj33', 'game', 2, 'jiangsu', 'nanjing', 5, 0, 0, 700000, 5);

insert into user\_access\_log\_aggr values('20171001', 'dasjffffksfj33', 'sport', 1, 'beijing', 'beijing', 8, 1, 0, 800000, 6);

insert into user\_access\_log\_aggr values('20171001', 'dasjdddksdfj33', 'sport', 2, 'jiangsu', 'nanjing', 20, 0, 1, 900000, 7);

insert into user\_access\_log\_aggr values('20171001', 'dasjeeeksdfj33', 'sport', 3, 'jiangsu', 'nanjing', 30, 1, 0, 600000, 10);

insert into user\_access\_log\_aggr values('20171001', 'dasrrrrksdfj33', 'news', 3, 'jiangsu', 'nanjing', 40, 0, 0, 600000, 12);

insert into user\_access\_log\_aggr values('20171001', 'dasjtttttdfj33', 'news', 4, 'shenzhen', 'shenzhen', 50, 0, 1, 500000, 4);

insert into user\_access\_log\_aggr values('20171001', 'dasjfkakkkfj33', 'game', 4, 'shenzhen', 'shenzhen', 20, 1, 0, 400000, 3);

insert into user\_access\_log\_aggr values('20171001', 'dasjyyyysdfj33', 'sport', 5, 'guangdong', 'guangzhou', 10, 0, 0, 300000, 1);

insert into user\_access\_log\_aggr values('20171001', 'dasjqqqksdfj33', 'news', 5, 'guangdong', 'guangzhou', 9, 0, 1, 200000, 2);

## 补充案例-小型流量分析系统：使用 logstash 将 mysql 数据导入到 Elasticsearch

1、安装es

adduser elasticsearch

passwd elasticsearch

chown -R elasticsearch /usr/local/elasticsearch

2、安装logstash-input-jdbc插件

yum install -y gem

gem sources --add https://ruby.taobao.org/ --remove https://rubygems.org/

gem sources -l

gem install bundler

bundle config mirror.https://rubygems.org https://ruby.taobao.org

在logstash目录下，vi Gemfile，修改 source 的值 为： "https://ruby.taobao.org"

在bin目录下，./logstash-plugin install logstash-input-jdbc

============================================================

wget https://github.com/logstash-plugins/logstash-input-jdbc/archive/v4.2.4.zip

unzip master.zip

cd logstash-input-jdbc-master

vi Gemfile，修改 source 的值 为： "https://ruby.taobao.org"

vi logstash-input-jdbc.gemspec

找到

s.files = `git ls-files`.split($\)

改为：

s.files = [".gitignore", "CHANGELOG.md", "Gemfile", "LICENSE", "NOTICE.TXT", "README.md", "Rakefile", "lib/logstash/inputs/jdbc.rb", "lib/logstash/plugin\_mixins/jdbc.rb", "logstash-input-jdbc.gemspec", "spec/inputs/jdbc\_spec.rb"]

gem build logstash-input-jdbc.gemspec

mv logstash-input-jdbc-4.2.4.gem /usr/local/logstash/

plugin install logstash-input-jdbc-4.2.4.gem

============================================================

3、启动logstash

在logstash目录中创建一份配置pipeline配置文件，user-access-log-pipeline.conf

input {

jdbc {

jdbc\_driver\_library => "/usr/local/mysql-connector-java-5.1.36-bin.jar"

jdbc\_driver\_class => "com.mysql.jdbc.Driver"

jdbc\_connection\_string => "jdbc:mysql://localhost:3306/website"

jdbc\_user => "root"

jdbc\_password => "root"

schedule => "\* \* \* \* \*"

statement => "SELECT \* from user\_access\_log\_aggr"

}

}

output {

elasticsearch {

hosts => [ "localhost:9200" ]

}

}

运行下面的命令检查配置文件语法是否正确：bin/logstash -f user-access-log-pipeline.conf --config.test\_and\_exit

启动logstash：bin/logstash -f user-access-log-pipeline.conf --config.reload.automatic

--config.reload.automatic，会自动重新加载配置文件的内容

## 补充案例-小型流量分析系统：安装和部署 kibana

1、下载kibana

从kibana官网下载，并且解压缩

./bin/kibana，即可运行kibana，用ctrl+c可以终止kibana进程

2、配置kibana

在$KIBANA\_HOME/config/kibana.yml文件中可以对kibana进行配置

server.port

server.host

elasticsearch.url

3、访问kibana web管理工作台

默认通过5601端口进行访问，通过浏览器访问即可

4、设置index pattern

我们需要为kibana配置一个index pattern来匹配es中的索引名称，默认是logstash-\*，匹配logstash写入es中的数据

同时还要配置一个time-field name，那个field是timestamp类型的，这是给kibana用来按照时间进行过滤的，kibana会自动加载出来给我们选择

## 补充案例-小型流量分析系统：基于 kibana 制作网站流量分析报表（一）

对指定的版块进行查询，然后统计出如下指标的汇总

pv: 所有人的pv相加

uv: 对userid进行去重

return\_visit\_uv: 回访uv

total\_visit\_time: 总访问时长

bounce\_visit\_uv: 跳出次数

curl -XGET 'http://localhost:9200/logstash-2017.10.14/logs/\_search?q=section:news&pretty' -d '

{

"size": 0,

"aggs": {

"pv": {"sum": {"field": "pv"}},

"uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}},

"total\_visit\_time": {"sum": {"field": "visit\_time"}},

"return\_visit\_uv": {

"filter": {"term": {"is\_return\_visit": 1}},

"aggs": {

"total\_return\_visit\_uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}}

}

},

"bounce\_visit\_uv": {

"filter": {"term": {"is\_bounce\_visit": 1}},

"aggs": {

"total\_bounce\_visit\_uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}}

}

}

}

}'

## 补充案例-小型流量分析系统：基于 kibana 制作网站流量分析报表（二）

对指定的版块进行查询，然后统计出如下指标的汇总

pv: 所有人的pv相加

uv: 对userid4进行去重

total\_visit\_time: 总访问时长

return\_visit\_uv: 回访uv

bounce\_visit\_uv: 跳出次数

curl -XGET 'http://localhost:9200/logstash-2017.10.14/logs/\_search?q=section:news&pretty' -d '

{

"size": 0,

"aggs": {

"pv": {"sum": {"field": "pv"}},

"uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}},

"total\_visit\_time": {"sum": {"field": "visit\_time"}},

"return\_visit\_uv": {

"filter": {"term": {"is\_return\_visit": 1}},

"aggs": {

"total\_return\_visit\_uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}}

}

},

"bounce\_visit\_uv": {

"filter": {"term": {"is\_bounce\_visit": 1}},

"aggs": {

"total\_bounce\_visit\_uv": {"cardinality": {"field": "userid", "precision\_threshold": 40000}}

}

}

}

}'